UNIVERSITY OF CALIFORNIA
Santa Barbara

# Compiler Estimation of Parallelism and Communication for Quantum Computation

A thesis submitted in partial satisfaction
of the requirements for the degree of

Masters of Science

in

Electrical and Computer Engineering

by

Jeffrey P Heckey

Committee in Charge:

Professor F. Chong, Chair

Professor T. Sherwood

Professor B. Parhami

September 2014

The thesis of
Jeffrey P Heckey is approved:

---

Professor T. Sherwood

---

Professor B. Parhami

---

Professor F. Chong, Committee Chairperson

August 2014

Compiler Estimation of Parallelism and Communication

for Quantum Computation



by

Jeffrey P Heckey

To my family for their undying support, my mother who irrationally believes in me, my uncle for familial, emotional, and ethanolic support, my two dads for encouraging me, my fellow researchers, without whom I would have never gotten this far or learned so much, my professors for engaging me, and to circumstances beyond my control for allowing me this opportunity.

# Acknowledgements

Fred Chong, Margaret Martonosi, Ken Brown have been instrumental in guiding and shaping the research presented here. Ali Javadi Abhari, Shruti Patil, Daniel Kudrow, and Adam Holmes are talented researchers, whose assistance and works have been invaluable.

# Abstract

## Compiler Estimation of Parallelism and Communication for Quantum Computation

Jeffrey P Heckey

Quantum computing promises to speed up scientific and computationally intensive operations. However, the power of quantum computing is limited by the relatively small window of time where the quantum state and be maintained (coherent). To achieve maximum efficiency, not merely to keep this state coherent but to increase computational productivity, maximizing the parallelism of the system is important. The architectural model that is explored here attempts to exploit the relatively small number of operations that are actually performed within a quantum computer to maximize fine-grained, data level parallelism, as opposed to the more common coarse-grained, task level parallelism. This model represents a Multi-SIMD processor design, where multiple SIMD cores are used to boost data level parallelism, but allows for limited task indepence.

The purpose of this work is to explore the effectiveness of parallel processing in a Multi-SIMD quantum architecture. It examines the ability to speedup computation using a combination of parallel processing scheduling and communication awareness, showing up to 7.8X speedup. This information is then used

to extract theoretical requirements for bandwidth (>8000 qubits/cycle peak) and throughput (3 qubits/cycle sustained). This research leverages the ScaffCC compiler toolchain [26], which provides a logical-level (i.e., implicitly error-corrected) quantum assembly output as the input to be scheduled and analyzed.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Quantum Computing (QC) has been known for decades for its potential to speed up algorithmically complicated computations [46]. QC has progressed to the point that certain classes of quantum computers are available commercially [1]. Many algorithms have been developed for quantum computers for such diverse applications as Fourier transforms and signal processing, database search, linear equation solvers, and molecular chemistry simulations [15, 19, 21, 60]. The algorithms typically allow for polynomial time implementations for algorithms with exponential runtimes in classical computers.

Of course, if it were easy it would be done by now. The main challenge to QC is coherence. The computer's state must be maintained in a state of quantum superposition throughout the duration of the computation: it must be coherent. When a computation is complete, precise measurements are made, collapsing this superposition, and providing a result for the algorithm. If the computer decoheres

during the computation, it is a bit like a power surge in a regular computer; if the system doesn't fail, an answer may be produced but may be incorrect.

Combating decoherence is done primarily two ways. The first is the application of quantum error correction control (QECC) to encode the data in a way that computations can be performed, but errors can be detected and corrected periodically [23, 53, 54, 55]. The second is architecturally, by structuring the design of the quantum computer so as to minimize the computation time. These two approaches buttress each other, with architectural design reducing the need for prohibitively large amounts of expensive error correction operations, and QECC providing a large amount of parallelism to exploit and improve the overall utilization of the system. Since QECC dominates both resources and runtime in large algorithms and datasets, the greater the reduction in runtime, the less QECC needs to be done, simplifying the whole system. The focus of this work is on the architectural approach. By exploiting parallelism in the algorithms at a fine-grained level and minimizing the communication overhead, this work intends to maximize the amount of computation that can be performed before the system decoheres.

Several types of quantum computers have been built [1, 29, 37]. Since 2003, ion-trap designs have been a lead contender in scalable quantum computer designs [3, 7, 11, 16, 27, 45, 59, 61, 62]. This design uses individual ions to encode

the physical quantum bits (qubits) and uses energy pulses from either lasers or microwaves to perform operations on these ions. Several architectures have been proposed for this technology [39, 58] that exploit its potential for parallelism.

This work seeks to reduce the runtime of benchmark quantum algorithms by examining the available parallelism within them, monitoring the communication requirements and eliminating unnecessary communication. As one of the preliminary surveys at this scale, determining potentially valuable avenues of research is also important. Some of the work presented here is also available reported in [22, 25, 26].

The research here leverages the efforts of the Scaffold team. Together we have built a compiler infrastructure that allows for a high-level, C-like description of the algorithm in the Scaffold language. The ScaffCC toolchain then compiles the algorithm and generates a gate-level view of the algorithm. These gates are the basis for reasoning about the architectural trade-offs between data-dependencies, parallelism, fault-tolerance overhead, communication overhead, execution time, and control constraints.

This work contributes to the growing body of quantum algorithm research by:

- extending the breadth and utility of ScaffCC to evaluate quantum architectures;

- being the first known effort to calculate the needs and trade-offs of communication in QC;
- proposing baseline algorithms to determine these values;
- comparative valuations of these scheduling algorithms;
- providing numerical values for communication throughput and bandwidth.

The remainder of this work is organized as follows: Chapter 2 discusses the basics of QC and the body of work being extended. Chapter 3 discusses specifics of the quantum algorithms evaluated, how they were chosen, and the technical details of the analysis. Chapter 4 and Chapter 7 discuss the analyses and their relationship and draws conclusions.

# Chapter 2

# Background

In order to understand the relevance of this work, it is helpful to have a baseline understanding of quantum computing principles. This section is intended to briefly describe how quantum computations are performed, the architectures targeted by this work, and the details of how the analyses are performed using the Scaffold language and ScaffCC toolchain.

## 2.1 Quantum Computation

The fundamental unit of data in quantum computer is the qubit. A qubit stores a single logical value and can be modified by a number of various gates or operations; this is similar to a classical computer. The difference is that the qubit is held in superposition, a probabilistic combination of 1 and 0. This is represented in Bra-Ket notation as $|\alpha\rangle + |\beta\rangle$, where $\alpha$ and $\beta$ are normalized complex numbers, such that $|\alpha|^2 + |\beta|^2 = 1$. This allows the quantum state to be modeled as a

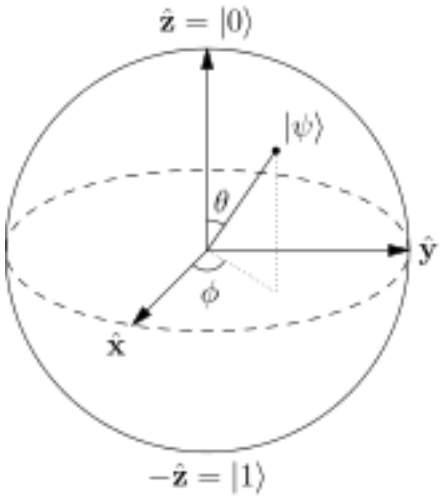


Figure 2.1: The Bloch Sphere is a mathematical representation of the possible states of a qubit or an $n$-qubit quantum system in $2^n$-dimensional space.

unit vector with an arbitrary phase. Both models can be extended to $n$ qubits by extending this vector to $2^n$-dimensional space. This vector can be mapped on to the Bloch Sphere (Figure 2.1), where any position on the surface of the sphere is a valid, unique superposition state for the quantum system.

Bloch Sphere rotations are norm-preserving, so the vector is always unit length. Rotations can be performed around any axis. In a typical quantum computer, common "gates" are defined with specific rotations around specific axes. Common rotation gates are X, Y, and Z (a $\pi$ radian rotation around their respective axis); S and T gates ($\frac{\pi}{2}$ and $\frac{\pi}{4}$ radian rotations) and their inverses, $S^\dagger$and $T^\dagger$ ($-\frac{\pi}{2}$ and $-\frac{\pi}{4}$). These gates are common in practice, though T gates in particular are expensive in terms of execution time and accuracy. The X gate is also called NOT because

it will swap the coefficients of $\alpha$ and $\beta$ in the Bra-Ket formulation. Two other gates make up the minimal set of gates required for quantum computation: the H (or Hadamard) gate is used to initialize a qubit to a superposition by giving both $|0\rangle$ and $|1\rangle$ equal probability, and the CNOT (or Controlled-X) gate is used to conditionally flip the state of one qubit based on the on the value of another. These nine gates describe the minimal universal of primitive gates and are the only gates used by Scaffold [26] (Section 2.4).

Gates in quantum systems are unlike logic gates in a traditional computer. While they have a physical location, any single location can implement any of the unitary (single qubit) operations, and adjoining gates can be used for CNOTs. The gate operation is instead controlled by external, energetic wave interactions, either electronic, magnetic, or photonic, which manipulate the quantum of state of the qubit within the gate.

More advanced gates exist that can be built out of this set. These gates include the Toffoli, a quantum NAND gate which is universal unto itself; the SWAP, which swaps the state of two qubits; and the Fredkin, a controlled SWAP [42]. All of the multi-qubit gates are reversible. This means that these gate have no fan-in or fanout, so a qubit must be supplied for each input and output. This is due to the *no-cloning theorem* [42], which says that the state of a single qubit cannot be copied to another qubit and be left unchanged itself. As qubits interact in these

gates, they become entangled, all qubits' states now being related to each other in some way.

These entanglements enable one of the more interesting aspects of quantum computing: teleportation. Entanglement allows for what Einstein called "spooky action at a distance" [5]: the ability for entangled particles to share information faster than light. By entangling two particles, their states are each a superposition of each other, such that observing the state of one instantly collapses the state of the other to a known state. Teleportation takes advantage of this by entangling two particles and then physically transferring one of the pair to the desired location. By entangling the qubit to be transported with a new temporary qubit, an *ancilla* qubit, and measuring both, the superposition can be recreated in the qubit at the destination. Teleportation is heavily used in quantum computing to move data quickly between locations (say a processor and memory) before the qubit's state has time to lose coherence.

Decoherence is the major obstacle to quantum computing. It puts a limit on the amount of time that can be used for computations before errors start creeping into the system. This can be counteracted by the use of *Quantum Error Correction* (QECC). By encoding logical qubits with redundant physical qubits, typically using Steane Codes [54], quantum computers can tolerate a certain error rate during processing allowing the computation to run indefinitely. The challenge

is that typical error correction overhead requires a two-level, recursive Steane code that encodes each logical qubit as 49 physical qubits and requires 23,409 timesteps after each normal computation timestep. As the amount of time spent doing error correction is so onerous, reducing a small amount of time spent on these computations has a large impact in overall performance.

There is one time that decoherence is good, though. When a qubit is measured it decoheres and loses its superposition, turning it into a classical 0 or 1 state. This measurement is how data is retrieved from the quantum computer. By measuring the final output qubits, the quantum wavefunction collapses to a deterministic state and an answer is available. However, since the superposition state of any qubit is described as the probability of being a 0 or 1, the final is not guaranteed to be the correct answer. Instead, a probable answer is returned and the computation is rerun multiple times to ensure certainty. Despite measurement causing a wavefunction to collapse and the attendant effects on entangled qubits, measurement is necessary for both teleportation and QECC while a computation is ongoing. The quantum superposition is preserved because each qubit is entangled with several others, so while the measured qubit state is now known other qubits in the system are still in superposition.

## 2.2 Quantum Technologies and Architectures

Many technologies can produce and control quantum computations: harmonic oscillators, optical photon, optical cavity, ion traps, nuclear magnetic resonance, etc. [42]; all of these have their strengths and trade-offs. The target of this research is ion traps since it appears to have the most promise of a realizable quantum system at this time. Ion traps can initialize and measure well characterized qubits with long decoherence times, and the technology has a universal set of high-precision, low error gates for performing calculation [16, 36, 40, 48]. The *DiVincenzo criteria* [14] requires all of these for realistic quantum computation. Experimental results have also shown the viability of all of the major features needed to construct a quantum computer with ion traps, including basic operation of moving, addressing, performing logical operations and storing qubits [63], entanglement [6], two-bit quantum gates [16], teleportation [47], and quantum error correction [9].

Typical ion trap designs will use lasers to operate on single ions (physical qubits) within the system. While that is functional, the lasers themselves require careful phase alignment and are physically large which presents significant difficulty in scaling the number of lasers that can used to control qubits. Some schemes have been proposed to use mirror arrays to allow for SIMD-style vec-

tor operations, where multiple qubits will go through the same operation with a single laser controlling all of them [29]. Another control strategy involves using microwaves, which could allow for a much larger qubit fanout [27, 50].

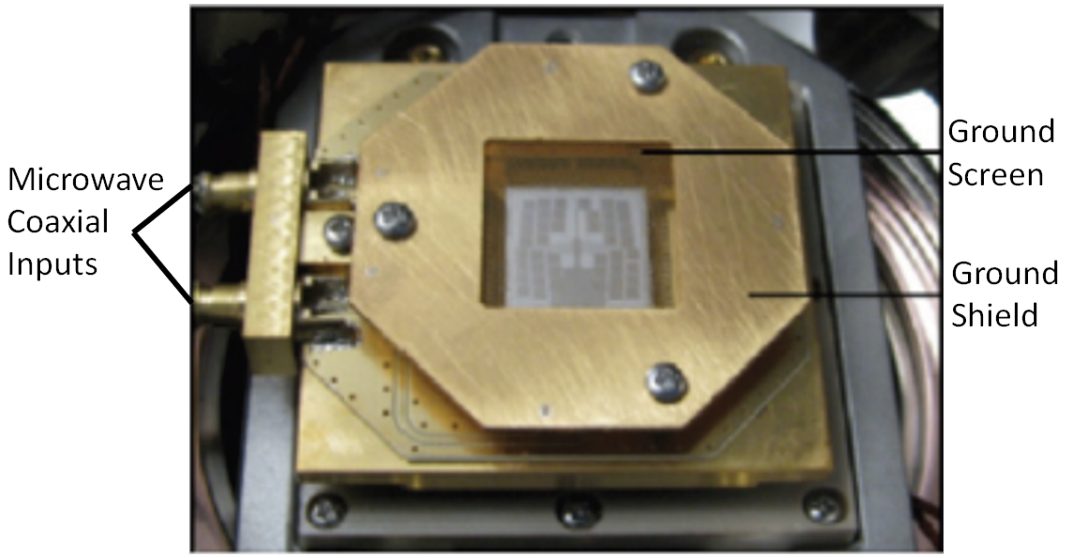


Figure 2.2: Microwave controlled ion trap. Installed device with microwave inputs. Reproduced with permission from [50].

Microwave controlled ion traps such as those shown in Figure 2.2 can scale much more easily than laser controls. An individual microwave signal can be used to control up to 100 qubits at a time [50], and splitting microwaves in a coaxial cable is much less technically difficult than controlling a mirror array for lasers. By fanning out the microwaves to 10 to 100 different arrays, up to 10,000 qubits may have the same operation performed on them simultaneously. This system could then be replicated to support multiple operations in parallel, allowing for operational level parallelism as well as data parallelism.

This architecture of having multiple discrete processing regions, each capable of executing a single operation on multiple qubits, is what enables the so-called Multi-SIMD($k$,$d$) quantum processor model that is the basis for this work. The $k$ value determines the number of computation regions in the system. This value is typically small (1-4) due to the limited parallelism found in the algorithms that are used (see Sections 3.2 and 3.3). The $d$ value refers to the number of qubits that can be operated on in a given SIMD region. This architecture also allows for unused SIMD regions to hold qubits for brief periods, essentially acting similarly to general purpose registers in a classical computer. This spatial property allows for reduced communication overhead, as shown in later sections.

By further incorporating ideas from *Compressed Quantum Logic Array* (CQLA) [58], whereby individual tiles are laid out to either be compute or memory, coherence within the system can be improved. Quantum computation suffers from a duality: computation requires qubits to be easily modified, but still be tolerant to noise and outside sources of error for long enough to do reasonable calculations. The idea behind CQLA is that compute regions can use smaller amounts of QECC while performing computations, while memory regions can use larger amounts of QECC to reduce the frequency with which error correction has to be performed while qubits are stored there. A global memory region is added to the 1-4 compute

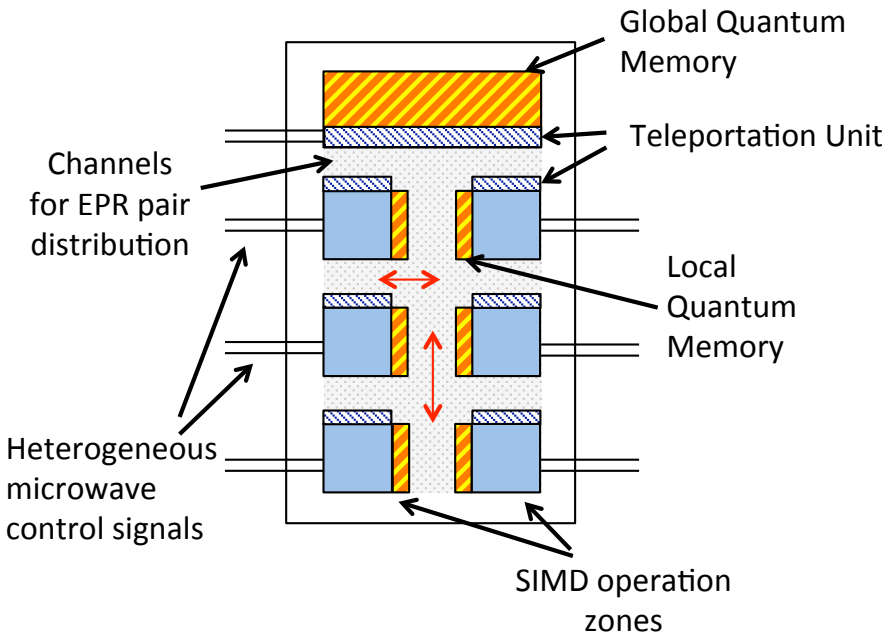


Figure 2.3: Block diagram of Multi-SIMD quantum architecture based on ion traps controlled by microwave technology. $k$ operating regions each support quantum operations on $d$ qubits simultaneously. Each operating region has a local memory for storing qubits. EPR qubit pairs are prepared near the global memory and distributed among regions via quantum teleportation.

SIMD regions above to store the logical qubits that will not be operated on in the current timestep. An example of this architecture is shown in Figure 2.3.

Note that the centralized global memory is a simplification that results from teleportation which makes the latency of long-distance communication constant (see Section 2.3). This constant global communication cost favors parallelism, since the latency of a single qubit move is the same as $d$ qubits, or even $k$ times $d$, because the operations needed to execute teleportation are the same, but the actual communication requires no physical transport channel. In fact, parallelism

breaks the computation into finer-grain chunks on separate regions. This reduces global memory accesses by leveraging local storage in regions and optional local memories, which is analogous to spatial computing approaches in classical computing [17, 56, 57].

Overall, efficient use of Multi-SIMD requires orchestration of qubits to maximize parallelism and to reduce qubit motion. Devising and evaluating effective scheduling techniques for Multi-SIMD is the focus of this work.

## 2.3 Data Movement and Teleportation

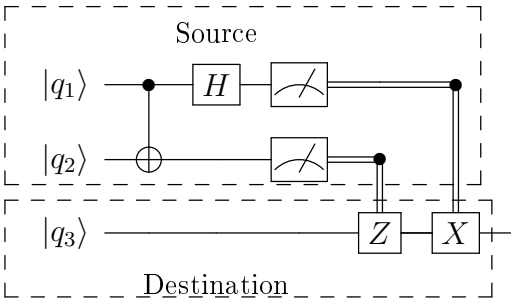


Figure 2.4: Communicating the state of $q_1$ using quantum teleportation: The EPR pair of $q_2/q_3$ is distributed prior to teleportaion, keeping $q_2$ near the communication source $q_1$, and using $q_3$ as the communication destination. By measuring the states of $q_1$ and $q_2$, and classically transmitting those measurement results (classical 0 and 1 bits), the state of $q_3$ takes on that of $q_1$ using none, one, or both of the X and Z operations at the target side. This completes the transmission of $q_1$ to $q_3$, while the state of $q_1$ is destroyed in the process.

At a physical level, communication in the Multi-SIMD($k$,$d$) architecture is assumed to be achieved through quantum teleportation, a phenomenon that makes

transmission of exact qubit states possible. Teleportation requires a pre-distribution of entangled Einstein-Podolsky-Rosen (EPR) pairs of qubits between the regions where communication will occur. EPR pairs are generated at the global memory, and distributed to the required regions. Figure 2.4 illustrates the computations required. The communication cost per move is four times as high as a single quantum gate operation; in a naive movement model, this quintuples the actual compute cost because of repeatedly moving qubits between SIMD regions and the global memory. In many cases the compiler can schedule teleportation operations in parallel with the computation steps.

In order to perform teleportation, EPR pairs must be distributed to each SIMD region and global memory so that the sender and receiver each have one half of the pair. The distribution of such EPR pairs has been studied in detail in [61]. Since repeated movement of qubits on the physical fabric is error-prone, teleportation reduces quantum decoherence by communicating information through a classical channel. While teleportation has constant latency with communication distance, longer distances do imply higher EPR bandwidth requirements (larger communication channels to move enough EPR pairs throughout the architecture). To minimize EPR bandwidth requirements, future work will investigate distributed global memory and compiler algorithms for mapping to such a non-uniform memory architecture.

## 2.4 Scaffold, CTQG, and QASM

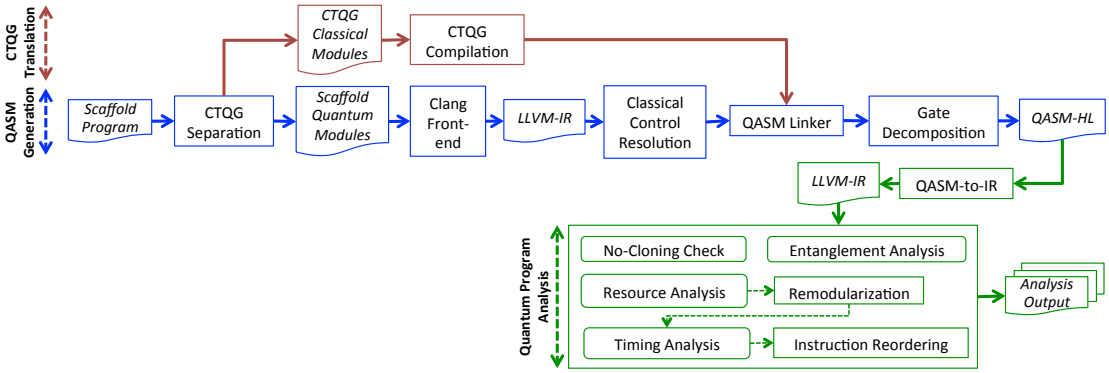


Figure 2.5: Data flow through the ScaffCC toolchain

The Scaffold and CTQG languages [25, 26] were developed to assist in the analysis of the quantum algorithms used in this project. Both languages are C-like in their basic form, but each has its own distinct syntax. A complete ScaffCC toolchain was built around the LLVM Compiler Framework [35], shown in Figure 2.5. This flow outputs QASM-HF, a quantum assembly language where all structure other than functional hierarchy is removed. Analyses of the compiled Scaffold code are performed based on the LLVM IR and QASM files for the results presented here.

The Scaffold language is a C-like language developed using the Clang frontend to LLVM. It allows a programmer to develop quantum algorithms in a familiar way, with the use of modules to describe reusable functionality and common flow control syntax like if-then-else, and loops. CTQG can be incorporated directly into

Scaffold files as modules and are separately compiled to QASM, but is processed through a different path in toolchain flow. While the LLVM compiler framework may be overkill for Scaffold's limited relatively small and uncomplicated feature set, it does provide a solid framework and libraries for handling compiler optimizations and code analysis. The major limitation is that all control structures in Scaffold must be classical, so while a particular set of gates could be told to iterate over all the qubits in a qubit array (a quantum register, or *qureg*), it can't determine the value of a qureg and loop over a qureg that number of times. This limitation is predominantly imposed by the superposition nature of the data – the loop would actually need to be performed with a probability distribution equal to the quantum superposition state of that qureg, which can't be done with discrete gates. This disconnect between the classical operation and quantum data results in a need for many operations that are typically conditional or recursive to be structured in fixed length and deterministic ways. Classical loops are typically handled by unrolling the loop at compile time; this is often coupled with aggressive constant propagation in order to determine the loop size.

CTQG (Classical-To-Quantum-Gates) alleviates some of the more challenging restrictions by allowing for arithmetic and conditional expressions with quregs. It can decompose assignment, basic addition, subtraction, multiply-accumulate, integer comparisons, and even some limited fixed point arithmetic and trigonometric

functions. Many of the simple arithmetic operations (addition, subtraction) can be performed using a recently developed algorithm by Cuccaro et al [12], with extensions provided for multiplication. CTQG does support limited conditional clauses (if-then-else) on quantum data through the use of CNOTs to create arbitrary *controlled unitaries*, operations that are only performed if certain conditions are met. CTQG is still limited to classically controlled loops; while loops, even over classical variables, do not exist; for loops over a range known at compile time will be unrolled into a sequence of gates. After the QASM was generated, it is reinserted into the QASM where the module was declared to run through the rest of the flow with the main code.

While the ScaffCC toolchain is largely a compiler, the bulk of the effort here has gone into the backend optimizers and analyzers. The first step in this research was to determine resource usage and determine the rough size of the algorithms proposed. This includes qubits, ancilla qubits, gates, and overall runtime. In order to achieve this optimization passes were made using static analysis of the program structure to find a purely gate level implementation. In order to remove control structures, which are entirely classical, all of the constants in the program need to be propagated through the program. For loops with defined bounds can simply be unrolled; this is only a matter of running the built-in LLVM loop unroller with a high enough threshold. While constant propagation is simple enough in a single

function, if a function is called multiple times with a constant argument, in a loop or even by different functions, the function's input arguments will be converted to constants and the function will be cloned with those new arguments hardcoded. The LLVM optimization framework made it easy to clone the functions and replace the original.

An additional pass was added to handle arbitrary rotation gates, Rx, Ry, and Rz. Often in quantum algorithms, a qubit will need to be rotated with arbitrary precision. A new optimization pass was added to find all of the rotation operations after constant propagation and use the Solvay-Kitaev algorithm [13, 33] to decompose the arbitrary rotations into discrete series of the gates described in Section 2.1. In order to perform the SK decomposition, the Single Qubit Circuit Toolkit (SQCT) [30, 31, 32, 52] was called as a modified static binary by the optimization pass. The decompositions were then inserted into the location of the rotation operation. Again, LLVM made it easy to modify the code quite effectively.

Once these optimization are complete, the program consists entirely of basic gates and modular hierarchy. The static code can then be analyzed for resource counts and scheduling purposes. Resources are analyzed by determining how many qubits are used within any function along with the number of ancilla qubits used. Scheduling is done by analyzing the data dependencies on the qubits and

establishing the necessary order of operations, then mapping to the resources allotted. Since scheduling is the main topic of this work, it is discussed more in the following sections.

Larger leaf modules provide more opportunities for good fine-grained scheduling, but when leaf modules are too large the scheduling time becomes unacceptably long. To find this balance, a flattening threshold ($FTh$) is chosen to increase the potential parallelism while keeping the scheduling time reasonable. The number of gates within each module, including submodules, are found by performing resource estimation analysis on them. If any module's size is less than the flattening threshold, then the module is flattened, i.e. all the function calls contained within it are inlined. This results in leaf modules that consist of at most $FTh$ operations.

The flattening threshold is determined by characterizing the initial modularity within a program. Figure 2.6 shows the percentage of modules with gate counts falling within specified ranges. This reveals the proportion of modules that could be flattened for a certain flattening threshold. Based on these and other experiments, the remainder of the analyses use $FTh$ set to 2 million operations. This flattened 80% or more of the modules contained within all benchmarks except *SHA-1*. For *SHA-1*, a flattening threshold of 3 million was used which flattened the entire benchmark.

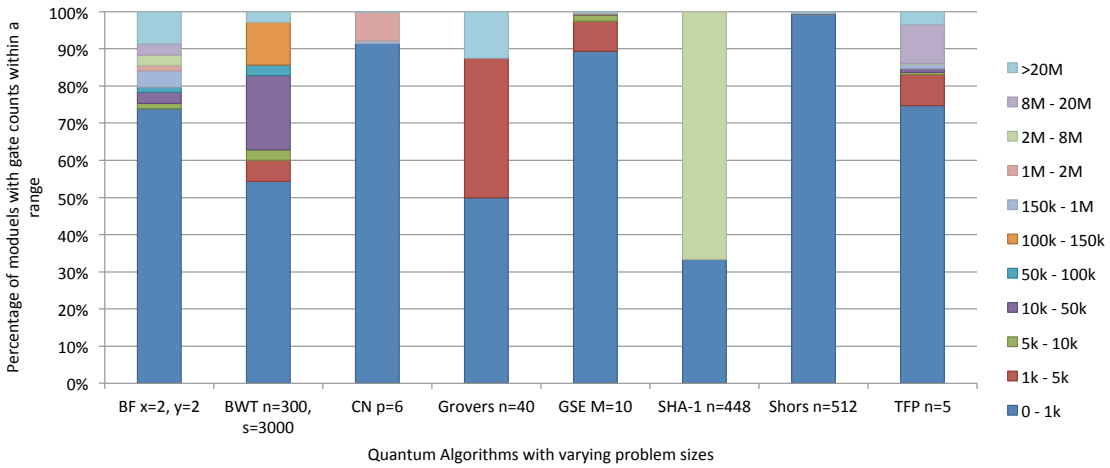


Figure 2.6: Histogram of gate counts represented as percentage of total modules in benchmarks. Using a flattening threshold of 2M operations, 80% or more modules were flattened for fine-grained scheduling for all benchmarks except SHA-1. For SHA-1, a flattening threshold of 3M was used to flatten the entire benchmark.

## 2.5 Execution Model

The Multi-SIMD($k$,$d$) model allows 1 to $k$ discrete, simultaneous operations to be executed in a single *logical* timestep, each of which can be applied on 1 to $d$ qubits apiece. For example, if 10 different qubits all require a CNOT operation applied to them, these can be positioned within a single operation region and the CNOT will be applied in a single timestep.

In the compiler all modules are blackboxes, so all active qubits are flushed to global memory during calls, since the position of all qubits are assumed to be in memory at the start of a module. This is mitigated by the fact that module calls are relatively infrequent and only cause a fixed overhead of a single teleportation

cycle. It is also assumed that all ancilla qubits are generated by the global memory and teleported to the SIMD region where they are needed.

Teleporting data between SIMD regions also must be orchestrated by the scheduler so the teleportation sub-operations shown in Figure 2.4 can be scheduled. If a qubit physically residing in one region is scheduled in a different region in the next timestep, it is moved to that region. If no operation is scheduled on that qubit and that region is active in the next timestep, it is moved to the global memory region.

The execution models and evaluations assume that each gate operation takes 1 timestep. Communication latencies are also accounted for, taking 4 timesteps. Some models of QC communication have a latency that varies proportionally to distance traveled, but QT approaches are distance insensitive. Rather, in QT, the bulk of the latency of each communication operation is the sequence of four qubit manipulation steps shown in Figure 2.4. Their schedule can be integrated into the computation schedule for the program. In order to simplify timestep sequencing and accounting, each timestep is constrained to the longest operational time, for example $10\mu$s for a CNOT [42].

# Chapter 3

# Methods

Computer architecture is about balancing the design requirements to find the optimal solution. By studying the communication costs of scheduling quantum algorithms in a Multi-SIMD architecture the tradeoffs inherent between different system resources, especially processing vs communication. While this has been explored extensively in classical computing arenas, this is the first work of an exploration at this level of detail for a quantum architecture. This work applies techniques pioneered in classical parallel computing to the new field of quantum computing.

The testing approach employed was to run all scheduling algorithms against a suite of quantum algorithms with various properties in terms of parallelism, data locality, length, levels of hierarchy, and classes of basic algorithms. The scheduling is performed with both communication costs and communication free metrics to demonstrate the true cost of communication within the algorithms.

Various scheduling parameters were used to influence the communication patterns within the schedules.

Data was processed on a desktop PC running Linux (Ubuntu 13.10). The data gathered used ScaffCC to compile and generate preliminary schedules, though an additional script was used to generate most of the schedules for leaf modules. All the leaf schedules were then scheduled hierarchically to create a complete algorithm schedule using the Scaffold backend coarse-grained scheduler.

## 3.1 Quantum Benchmark Algorithms Used

The benchmark algorithms were developed by several researchers over about three years as a part of the Scaffold effort. Specifications were provided by IARPA as benchmarks for demonstrating the effectiveness of different quantum compiler toolchains, of which ScaffCC was one. These benchmarks address many problems in computer science: factorization, search, eigenvalue estimation, phase estimation, discrete logarithms, and order and period finding [41]. They are currently the largest and most sophisticated quantum algorithms developed known. Each benchmark is given and described below.

- Shor's Factoring Algorithm (SF): The classic example of a quantum algorithm, the first killer app. It performs factorization using the quantum

Fourier transform and operates in polynomial time (instead of exponential time in classical computers) [51]. The problem size is parameterized by $n$, the size in bits of the number to factor.

- Grover's Search Algorithm (GS): Uses a quantum concept called amplitude amplification to search a database of $2n$ elements [19]. The problem size is parameterized by $n$.

- Binary Welded Tree Algorithm (BWT): Uses quantum random walk algorithm to find a path between an entry and exit node of a binary welded tree [10]. The problem size is parameterized by height of the tree ($n$) and a time parameter ($s$) within which to find the solution.

- Ground State Estimation (GSE): Uses quantum phase estimation algorithm to estimate the ground state energy of a molecule [60]. The problem size is parameterized by the size of the molecule in terms of its molecular weight ($M$).

- Triangle Finding Problem (TFP): Finds a triangle within a dense, undirected graph [38]. The problem size is parameterized by the number of nodes n in the graph.

- Boolean Formula (BF): Uses the quantum algorithm described in [4], to compute a winning strategy for the game of Hex. The problem size is parameterized by size of the Hex board ($x$, $y$).

- Class Number (CN). A problem from computational algebraic number theory, to compute the class group of a real quadratic number field [20]. The problem size is parametrized by $p$, the number of digits after the radix point for floating point numbers used in computation.

- SHA-1: A quantum implementation of the classical Secure Hash Algorithm 1 [43]. The problem size is parameterized by the size of the message in bits ($n$).

Each benchmark has two problem sizes that were chosen based on known complexity increases within the algorithm (such as SHA1 iterations) or the limits of the computing hardware running the scheduling (some algorithms, notably TFP and CN, required special processing in order to complete).

## 3.2 Ready Critical Path (RCP) Algorithm

The Ready Critical Path (RCP) algorithm was modified from the algorithm of the same name in [64, 65]. The algorithm is designed to both minimize the processing time by allowing all (or as many as possible) of its dependencies to

be already available and to minimize the communications by assigning the task to the processor where the most data is available, reducing communication costs. This scheduling is greedy in that the decisions are made based on the current state of the system without considering future states. In this Multi-SIMD architecture, the data dependency issues are equivalent between "free" and "ready" tasks due to the fact that every operation is only a single cycle and can only be free or ready when all of its dependencies are satisfied. The secondary concern, data locality, is of paramount importance, especially when the communication cost is felt so acutely as in this architecture (up to 80% of the total runtime).

This implementation of RCP bases its prioritization on three factors: the operation type, the communication costs, and the data dependency *slack*. Since the architecture uses SIMD execution, the occurrence of an operation type is positively correlated with the priority; so if there are 30 CNOT operations and only two H operations (and all other factors are equivalent), the CNOTs will be scheduled first. The distance priority is calculated by determining if data needs to move between the SIMD regions or memory. If a qubit is already in a given SIMD region, it will be more inclined to stay in place, limiting movement in the system. Additionally, a factor called slack is negatively correlated with priority; slack is the graph interval between uses of a given qubit (in the case of multiple qubits, the slack is taken as the minimum distance). This slack factor addresses potential

greedy scheduling issues by deprioritizing operations that don't require immediate execution. Slack is decremented at each timestep (to a minimum of 0) in order to increase urgency over time, preventing starvation or deadlock scenarios. The priority for each operation is then summed as the priority for a given operation type for each SIMD region. The highest weight for each SIMD region is then chosen and scheduled; SIMD regions may be scheduled out of numerical order based on these priorities. the RCP algorithm is shown in Algorithm 1.

**Algorithm 1** The RCP algorithm. At each timestep, it computes the relative weight for scheduling an operation type to each SIMD region based on the prevelence of that operation, the distance of each qubit from that SIMD region, and the graph distance to the next use of that qubit. The highest weighted operation type is then scheduled to its preferred SIMD region and the calculation is repeated until no more operations can be scheduled. The getSimdOp function is in Algorithm 2 and updateRcpQ is in Algorithm 3.

```
Function rcp(Module my_module) is
    int simd, ts = 0;
    OPTYPE optype;
    RCP schedule;
    OP rcpq[] = my_module.top();
    while not rcpq.empty() do
        int simds[] = 1..k; while not (simds.empty() or rcpq.empty()) do
            {simd, optype} = getSimdOp(rcpq, simds);
            schedule[ts][simd] = rcpq.pop_all(it.optype);
            simds.delete(simds.find(simd));
        end
        updateRcpQ(ts, rcpq); ts++;
    end
end
```

**Algorithm 2** The getSimdOp function calculates the priorities for each option in the ready list given the available SIMD regions and returns the optype and the SIMD region to assign it.

```
Function getSimdOp(OP rcpq, list of int simds) : {int, OPTYPE} is
    for each op in rcpq do
        optypeCnt[op.optype()]++
        for each qubit in op.args() do
            locs[op] |= 1 << qubit.simd();
        end
        // foreach location
    end
    // foreach op for each simd in simds do
        for each op in rcpq do
            weight = O * op.optype() + D * (locs[op] & (1 << simd)) - S *
            op.slack(); if weight > max then
                max = weight;
                maxSimd = simd;
                maxOptype = op.optype();

            end
        end
    end
    return {maxSimd, maxOptype};
end
```

**Algorithm 3** The updateRcpQ function adds all newly available operations into the RCP ready queue.

```
Function updateRcpQ(int ts, OP rcpq[], RCP schedule) is
    for each op in schedule[ts] do
        for each child in op.children() do
            if child.ready() then
                rcpq.push_unique(child);
            end
        end
        op.slack–;
    end
end
```

## 3.3 Longest Path First Scheduling (LPFS) Algorithm

The Longest Path First Scheduling algorithm is novel to the best of the author's knowledge. It was developed based on the idea that long serial paths within most of the algorithms here would allow qubits to remain in a SIMD region longer, and thus reduce the communication overhead. The algorithm takes the DAG of a leaf module and allocates one longest path to each of a subset of the SIMD regions available. Once all of the allocated SIMD regions are assigned, the remaining SIMD regions are used to schedule any other operations that are not on that path. These operations are pulled from a ready list, which only stores operations that have all of their dependencies already fulfilled so no conflicts can arise. Since many operations are single qubits, they will be repeatedly operated on without intervening qubits moving in and out. Any CNOT operations will require the single missing operand to move in and out, allowing for a natural overlap in data dependencies; if the operands can be used before or after the CNOT as well the moves are reduced further.

One of the challenges of this algorithm is that most longest paths will be of unequal lengths. If one path is significantly shorter than the next longest, that SIMD region will be idle until the module's schedule is complete. There are two

possibilities: open the allocated region to opportunistic scheduling from the ready list, or find another long path to insert into the unused region. This choice is controlled by the "refill" option provided to the algorithm.

Another challenge is that the algorithm as described only allows for a single operation to be performed per SIMD region. By filling any SIMD region with as many operations from the ready list as possible, greater computational efficiency can be achieved, though most likely at the cost of communication overhead.

## 3.4 Hierarchical Scheduling

To allow benchmarks to scale beyond tractable sizes (beyond a few million operations), a coarse-grained scheduler stitches together optimized schedules for leaf modules scheduled by RCP and LPFS in the call tree. The scheduler uses a simple, list-based approach to schedule leaf modules and operations in non-leaf modules with the goals of improving parallelism and/or reducing communication overheads.

Given a set of pre-scheduled leaf nodes, the coarse-grained scheduler completes the program scheduling by using list scheduling to compose together a full schedule. Operations are assigned priorities based on criticality and scheduled in priority order. The quantum gate operations in non-leaf modules are scheduled

along with invocations to other modules. The invoked leaf modules have been previously scheduled by one of the fine-grained schedulers discussed next and are now treated as blackbox functions. Once the schedule for a module is determined, it is characterized as a blackbox with a length dimension equal to schedule length, and a width dimension equal to highest degree of parallelism found in the schedule.

To allow the coarse-grained scheduler to effectively parallelize the invoked blackboxes within the width $k$ constraint, flexible rectangular dimensions are used for each blackbox. During fine-grained scheduling of each module, multiple schedules are determined to find schedule lengths with SIMD widths between 1 to $k$. The coarse-grained scheduler is presented these blackboxes with multiple dimensions. When parallelizable modules are encountered, the combination of blackboxes that yields the minimal length subject to the width constraint is chosen. Algorithm 6 shows the pseudo-code for coarse-grained scheduling with flexible blackbox dimensions.

In a $k$-resource constrained schedule, the width for any invoked module is at most $k$. Any operations (other than invoked modules) encountered by the coarse-grained scheduler have an operation execution cost of 1 and a movement cost of 4.

## 3.5 Algorithm Optimizations

For the purposes of rapid early development the algorithms were initially written in Perl, while intending to eventually port the more developed algorithms back into the ScaffCC LLVM flow in C++. Due to time constraints and continued modifications to the basic code, the Perl code was never ported to C++. This became a challenge in terms of the overall runtime, with some medium-sized algorithm configurations taking upwards of 16 hours to run. This was obviously impractical for the number of test cases that needed to be run. Several modifications were made using various techniques and tools to speed up the run time. These optimizations are discussed in this section.

The longest path finding algorithm for LPFS was extremely slow because of repeatedly searching and heavy memory usage. An early version of this algorithm started at the top of the DAG (or an intermediate ready list) and would descend to each node's child and would iterate through the tree that way. This resulted in a huge recursive fanout for each next step, recalculating and following the children, often repeatedly. This was not obvious from inspection in the code so a Perl profiler, NYTProf [34], was used to isolate the highest frequency code blocks. Initially, the getNextLongestPath() subroutine was taking obviously the largest amount of time. The first fix was to reduce the calls to it by setting a flag to

prevent calling it once it was unable to find anymore paths. This helped, but only minimally. The profiler indicated that a subroutine call to ::max(), which returns the maximum value in an array, was taking the most time in getNextLongestPath() so this was turned into a local comparison between the two variables being checked, which saved about 2 seconds out of an 11 second runtime on a short scheduling problem, or about 18%.

All of those updates were small, incremental improvements. At this point the profiler was abandoned for more direct inspection of the actual data flow through getNextLongestPath(). A few debugging variables were added to count the number of nodes and their children that were analyzed. The surprising result was that getNextLongestPath() was apparently exploring $O(n^2)$ nodes. The realization was that as node's children were analyzed, they were added to the next iteration list. While they were added uniquely, any fanout would double the number of paths followed by this list. It was apparent that a single pass of all of the nodes should be sufficient. In order to prevent corruption of the depth calculation it was only necessary to ensure that all parents were scanned before their children. When new operation nodes are created, they are automatically scheduled in ASAP order to determine critical path and maximum potential parallelism width. By walking this schedule the children's depth could be computed correctly definitively. No measurements were taken at this time, but runtimes were drastically reduced. An

additional enhancement was made to simply walk the list of operations for a given module, avoiding dereferencing the ASAP information; since the graph is built in operation order, children always come after parents. This was shown to decrease the system runtime 33% for small workloads.

Returning to the profiler, the update_ready() function that was created and initially shared between LPFS and RCP was inspected. This function would take the operations that had just been scheduled at the current timestep and would then determine which children were ready to be scheduled in the next timestep. After profiling it became clear that this was one of the largest contributors to runtime, though it was initially dismissed as a necessary step. The most challenging part of this algorithm was ensuring that only one instance of any operation was in the list. Traditionally this done with an insertion sort technique that runs in roughly O(n) time when individual elements are being added (as opposed to full sorting). However, Perl has a native hash data type that uses a key:value pair, so keys are inherently unique. This was used internally to update_ready(), but by promoting it to lpfs() many of the individual sorts (O(n log n)) could be turned into hash accesses (O(1)). The one time that the ready list needs to be accessed in an ordered fashion is to find the next operation to schedule in an unassigned SIMD region; to ensure that a certain uniformity is seen between runs, this is chosen as the operation with the lowest ID. This access requires performing a

search on the keys (the operation IDs) which is a simple integer comparison for the minimum given the list of keys and is done in exactly $n$. Due to the reduced amount of sorting based on object values and the direct access provided by the hash, the runtime was cut by about 3% on small workloads. In order to prevent breaking functionality, update_ready() was duplicated specifically for LPFS while RCP was not being used.

The function update_moves(), which calculates the qubit movements between timesteps, was the next target. This function initially moved qubits out of SIMD regions that were inactive and would move them back in later with no operations being performed. Seeing that holding the qubits in the SIMD region during a no-op should not cause issues, it was determined that this should instead scan back to the last active timestep for SIMD region and determine if the qubit should now be moved to memory. This searching was problematic because it incurred heavy pointer indirection and required iterating back over the tree. A new field was added to the Schedule object to store the names and corresponding SIMD regions of all active qubits. This was compared with the qubits that had been assigned to the current timestep. Both of these were then used to calculate which qubits needed to stay in place (no action), move between SIMD regions, or be fetched from or stored to memory. This resulted in a 28% speedup for this function. When coupled with additional enhancements (removing additional calls to ::max(),

defining local variables to prevent pointer indirection) resulted in an additional 11% speed up over a moderate workload.

For all of these optimizations, there were still some workloads that were running for several hours, sometimes many days. Given the long runtimes and efforts to reduce the overall runtimes, the Unix utility "time" would be used to monitor the regressions. Regressions would frequently be run in parallel on different data sets since the program is single threaded and greater utility could be derived from the machine. Surprisingly many long runs would still report relatively small system time durations. A final solution to this was to simply set the "nice" level of the regressions to -10, allowing them to run with more aggressive scheduling. This drastically reduced the overall runtime, several runs that would occasionally take 6 hours would now be cut down to a few minutes. The best guess is that with higher scheduling priority the memory swapping is reduced in the system; this could also be exacerbated by running several instances in parallel. After getting some of the longest runtimes down from 6 hours into the 6-12 minute range, the need for parallelism was also reduced, further decreasing memory utilization and virtual memory paging.

**Algorithm 4** The Longest Path First Scheduling algorithm. The $l$ paths are scheduled to allocated SIMD regions, all other operations go in other regions. SIMD operation and reallocation are possible with the "SIMD" and "refill" options.

```
Function lpfs(DAG G, list of int simds, int l) : Schedule S is
    for i in 0 to l-1 do
        // Get longest paths for allocated SIMD regions
        simd[i] = getNextLongestPath(G.top);
    end
    for op in G.top do
        // Initialize ready list
        if (! op.followed) then
            ready.push(op);
    end
    ready = G.top();
    while (! ready.empty() && ! simd.forall().empty() ) do
        // Schedule each time
        for i in 0 to l-1 do
            // Schedule allocated SIMD regions
            if (refill && simd[i].empty()) then
                // Reuse SIMD region if it is out of operations
                simd[i] = getNextLongestPath(ready);
            end
            op = simd[i].pop();
            S[time][i].push(op);
            if (opportunistic_simd) then
                // Schedule ready operations of the same type
                S[time][i].push(ready.getAllOps(op.op_type));
            end
        end
        for i in l to k-1 do
            // Schedule unallocated SIMD regions
            optype = ready.top().op_type;
            S[time][i].push(ready.getAllOps(op.op_type));
        end
        for op in S[time].forall().getAllOps() do
            // Update ready list
            ready.push(op.getReadyChildren());
        end
        ready.uniq();
        time++;
    end
    return S;
end
```

**Algorithm 5** Get Next Longest Path algorithm. This function takes a list of ready operations and walks their children until they reach the end of the module and finds the one with the longest distance. The longest distance is backtraced until it reaches the start, recording each operation in the path and returning the path. Each operation is marked as followed so subsequent calls can avoid taken paths.

```
Function getNextLongestPath(list of Op ready) : list of Op is
    Op path[];
    Op last = ready.top();
    // Reset distances of all untaken paths
    for op in ready do
        if (! op.followed) then
            op.dist = 1;
        end
    end
    // Search thru schedule for longest path
    start_level = MAX_INT;
    for op in ready do
        for child in op.children do
            if (! child.followed) then
                child.dist = max(child.dist, op.dist + 1);
            end
            if (child.dist > last.dist) then
                last = child;
            end
        end
    end
    // Backtrace path
    path.push0(last);
    while (path[0].dist > 1) do
        for parent in path[0].parents do
            if (parent.dist == path[0].dist-1) then
                path.push0(parent);
                parent.followed = 1;
                break;
            end
        end
    end
    return path;
end
```

**Algorithm 6** Hierarchical scheduling algorithm for $k$ SIMD regions. Operations are scheduled in priority order similar to list scheduling. Flexible blackbox dimensions are considered for parallelizable modules to find the best combination for them.

```
for each non-flat module do
    //Track schedule in terms of blackbox dimensions
    totalL = 0; totalW = 0; // total length and width
    currL = 0; currW = 0; //current length and width
    for each operation F_i in a priority-ordered set of operations {F: Priority(F_i) ≥ Priority(F_j) if (i < j)} do
        Check predecessors to find the earliest timestep t_e in which F_i can be scheduled
        Get width W and length L for F_i
        if (t_e ≤ totalL + currL) then
            // dependencies show that F_i can be parallelized with previous schedule
            if (currW + W ≤ K) then
                //parallelize the operation F_i
                timestep(F_i) = max(totalL+1, t_e)
                currW = currW+W
                currL = max(currL, timestep(F_i)+L)
                F_p = {F_p,F_i} //Add to set of parallel functions in current schedule
            else
                //k-constraint would be violated if parallelized
                for set of functions {F_p,F_i} do
                    Try all combinations of possible widths, and compute length.
                end
                if one or more combinations found with combined width ≤ K then
                    Choose combination with smallest length.
                    currW = Width of combination
                    currL = Length of combination
                    F_p = {F_p,F_i} //Add to set of parallel functions in current schedule
                else
                    //serialize F_i due to k-constraint
                    totalW = max(totalW, currW)
                    totalL = totalL + currL
                    timestep(F_i) = totalL+1
                    currW = W; currL = L
                    F_p = {F_i} //set of parallel functions in current schedule
                end
            end
        else
            // serialize F_i due to data dependency
            totalW = max(totalW, currW)
            totalL = totalL + currL
            timestep(F_i) = totalL+1
            currW = W; currL = L
            F_p = {F_i} //set of parallel functions in current schedule
        end
    end
    //merge current box with total box dimensions
    totalW = max(totalW, currW)
    totalL = totalL + currL
    Store totalW and totalL in data structure
end
```

# Chapter 4

# Results

The primary goals of this work are to determine the architectural costs of quantum benchmark algorithms and how to optimize these benchmarks using different scheduling algorithms. To that end, eight benchmark algorithms were chosen to evaluate both the architectural configurations and the various scheduling algorithms that have been discussed. Results were generated by evaluating many different configurations of the benchmarks, architecture and scheduling algorithms. By carefully varying the different parameters a comparative understanding of the different configurations are possible.

All benchmarks are run with two problem sizes chosen to demonstrate how the benchmark scales. The architecture itself has two parameters, Multi-SIMD($k$, $d$), which control the number ($k$) and size ($d$) of the SIMD execution regions. Typically $k$ is either 2 or 4, given the low amount of parallelism in many benchmarks, except where otherwise noted. Likewise, most of the data presented here uses a $d$

value of 1024. The RCP algorithm has three weighting parameters: $O$, $D$, and $S$; typically all are set to one (equal weight), but other configurations are examined to see what weighting is most effective. The LPFS algorithm requires at least a single SIMD region to run, so the $l$ parameter was set to 1, except where otherwise noted; additionally, options for opportunistic SIMD scheduling ("SIMD") and longest path refilling ("Refill") were used, except where otherwise noted.

In this section results are presented for the following:

- Runtime speedup of instruction- and data-level parallelism
- Runtime speedup with data movement analysis
  - Data-parallelism sensitivity
  - RCP configuration variability
  - LPFS configuration variability
- Communication costs, requirements and limits
  - Sustained throughput requirements
  - Peak bandwidth limits

## 4.1 Runtime Speedup of Instruction- and Data-Level Parallelism

The first set of results, in Figure 4.1, show the comparison of the speedups provided by a Multi-SIMD architecture and the theoretical maximum speedup along the critical path, looking solely at the instruction and data parallelism within the benchmarks found by the RCP and LPFS algorithms.

All of the scheduling algorithms except Shor's n=512 were able to achieve near-theoretical Critical Path speedup at either $k = 2$ or 4. This is assisted by using $d = 1024$, which allows for clustering as many qubits as possible into SIMD regions. Additionally, RCP speedups are lower than or equal to LPFS in every benchmark except TFP n=5 at $k = 2$.

Shor's algorithm shows a greater sensitivity to the number of SIMD regions available $k$. The parallelism of Shor's algorithm with higher $k$ is shown in Figure 4.2. The reason for this can be traced to the large number of rotation operations that exists in this code. These rotation operations can theoretically execute at the same time because they are on distinct qubits, except for the fact that practically they need to be decomposed into primitive, standard operations (as described in 2.4). This can prohibit the parallelization of operations unless more SIMD regions are created to accommodate them, as illustrated in Table. 4.1. Since

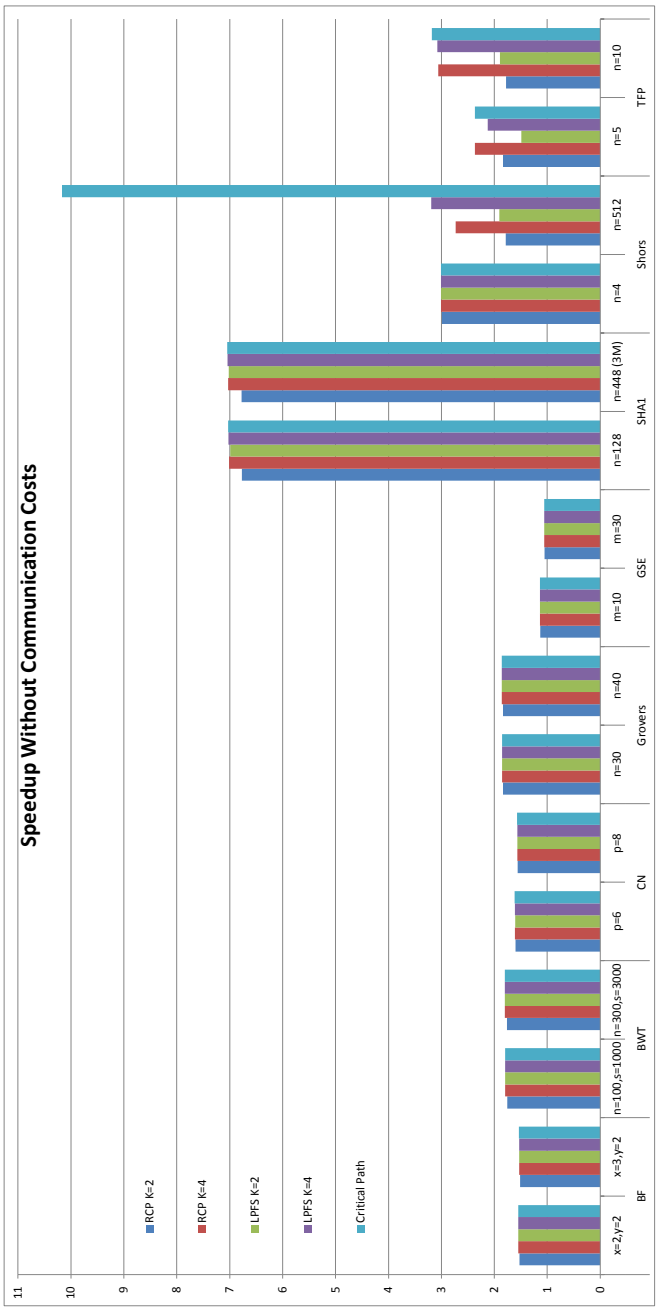


Figure 4.1: The speedup over sequential execution of each benchmark with each scheduling algorithm, compared to the estimated critical path. Almost all algorithms, except Shor's, achieve near-complete speedup by $k = 4$.

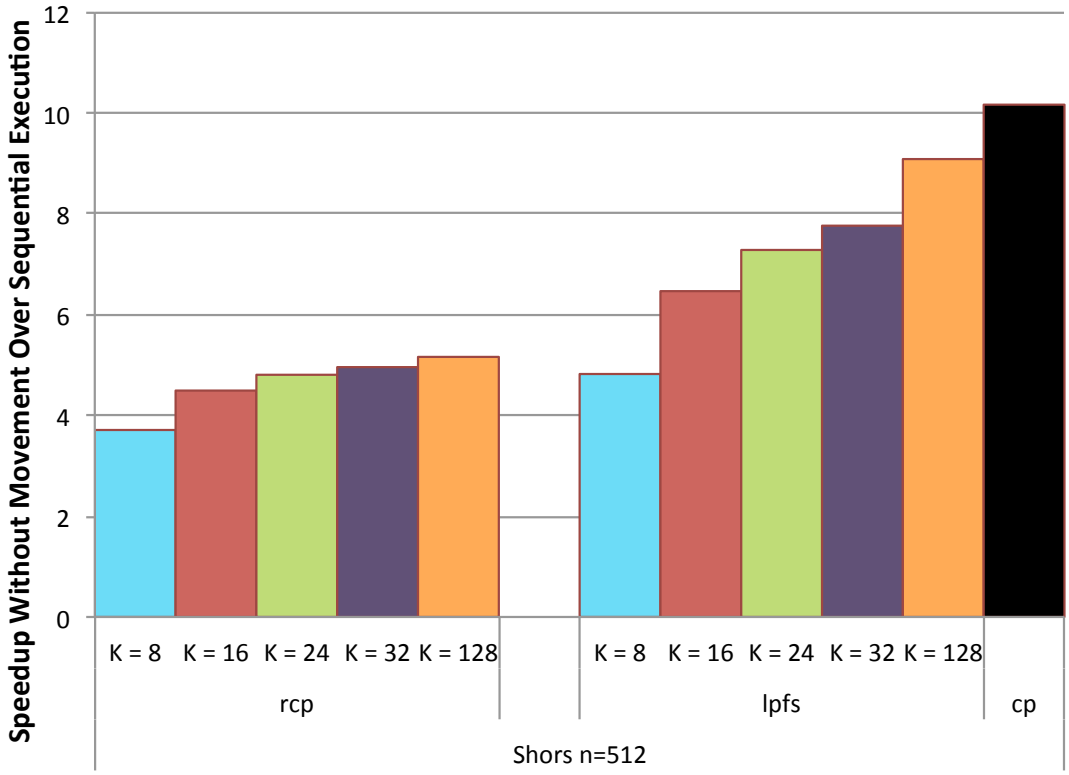


Figure 4.2: Shor's algorithm speedups as scheduled with a communication-aware scheduler on a Multi-SIMD architecture with local memories. High numbers of rotations cause long serial threads of operations to each execute on a separate SIMD region, thus getting better gains with higher $k$.

many of these rotations were not inlined into the code, to keep the size manageable, they remain as blackboxes in the course-grained schedule. That causes the scheduler to allocate a separate region to each, effectively increasing the need for these regions. Improved hierarchical scheduling would make better use of the $d$ parallelism.

TFP's improved RCP performance is also because of coarse-grained scheduling. LPFS requires $l$ SIMD regions to be used for the longest $l$ paths, so all leaf schedules are force to have a width of $k = 2$; RCP allows for $k = 1$ widths. Since the TFP algorithm also has several rotation decompositions that only operate on a single qubit, more of these rotations can be scheduled in parallel using the coarse-grained, flexible boundary scheduler with RCP resulting in a shorter overall runtime.

| Rotation Operation | Primitive Operations Approximating Rotations |
|---|---|
| $R_z(q_1, \theta_1)$ | $T(q_1) - S^\dagger(q_1) - H(q_1) - Z(q_1) - ...$ |
| $R_z(q_2, \theta_2)$ | $H(q_2) - Y(q_2) - X(q_2) - H(q_2) - ...$ |
| ... | ... |
| $R_z(q_n, \theta_n)$ | $S(q_n) - X(q_n) - T(q_n) - T^\dagger(q_n) - ...$ |

Table 4.1: Parallel rotations cannot be executed simultaneously on a hardware with primitive operations, unless there are enough SIMD regions to accommodate them.

## 4.2 Runtime Speedup with Data Movement Analysis

Figure 4.3 shows all scheduling algorithm speedups over a naive movement model where data is moved between SIMD regions and global memory every timestep, effectively increasing the overall runtime by 5X (1 timestep for the operation, 4 timesteps for the communication). All algorithms show some speedup over communication-unaware runtime models due to reduced movement. The critical path was not used for a theoretical bound in these or the next results because no suitable critical path model for incorporating movement was found. An average increase in speedup of 57% is seen across all algorithms. The largest gains are seen in GSE (307%) and Shor's (209%).

Some algorithms, such as BF, CN, Grovers, and SHA-1, all have a large number of highly dependent, serial operations. BF, CN and SHA-1 are composed of several CTQG modules, which produces unoptimized code that is highly locally serialized. This results in benchmarks that have a low degree of parallelism and aren't well optimized. The interactions between data dependencies also result in many small (1-2 qubit) moves between global memory and various SIMD regions that can't be skipped to improve performance.

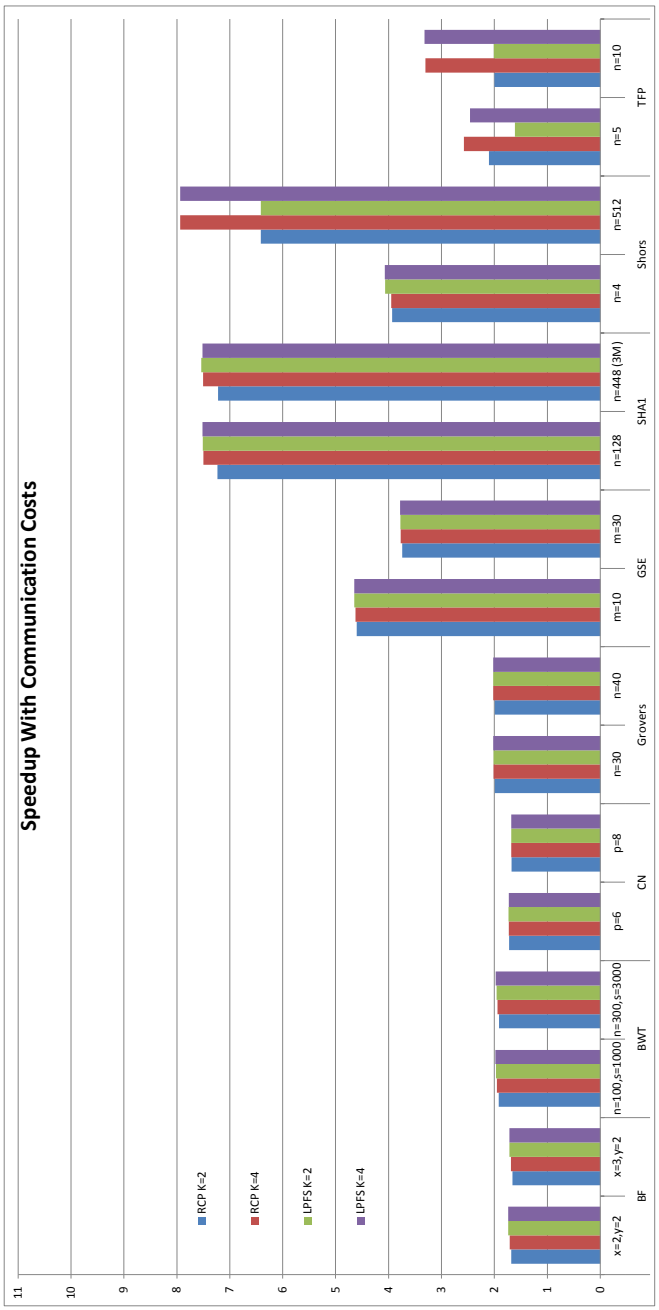


Figure 4.3: The speedups using a communication-aware scheduler over a sequential, naive movement model. All benchmarks show improvement over Fig. 4.1, with GSE and Shor's showing the largest gains.

GSE shows the largest gains due to its distinctive structure. Two key qubit registers containing the primary active qubits are rarely moved out of an SIMD region once they are in place and typically have long sequences of operations on the same qubits. This results in very few moves either between SIMD regions or memory.

Shor's algorithm also consists of fairly serial modules derived from CTQG and lots of rotations which need to be decomposed, which results in a large number of moves being removed from those modules. However, it is likely that with greater data parallelism (by not blackboxing the sub-modules or through further flattening), the speedup could increase while communication increases because the efficiency of the schedule would be higher.

### 4.2.1 Data-parallelism Sensitivity

Though $d$ is constrained to 1024 for the majority of the results presented, a subset of benchmarks with $d$ scaling is shown in Figure 4.4. GSE was chosen because of its structure and has a high degree of data parallelism. Here, $\infty$ is simply sufficiently high to allow the maximum data width, or around 64K. SHA-1 and Shor's algorithms are both heavily data parallel algorithms traditionally, so they were also chosen to demonstrate the extent of $d$ scaling. As shown in the extreme case of $d = 2$, SHA-1's high data-parallelism is significantly impacted by

the reduction, but the much more task parallel Shor's is not; GSE takes a moderate hit, indicating that while it is reliant on data-parallelism, the movement factor is much more important. The overall speedup between $d = 128$ and $\infty$ is rarely significant. As there is little gained beyond a certain level of data parallelism, it is largely omitted from analysis here.

### 4.2.2 RCP Configuration Variability

Figure 4.5 shows the speedups of RCP at $k = 2$ and 4 with different parameter configurations. RCP has three configurable parameters: $O$ is the weight for operation type, $D$ is the weight for move distance, and $S$ is a negative weighting for graph distance until the operation is needed. RCP, as a traditional scheduling algorithm, is meant for tasks of varying length and full threads of execution. The fine-grained scheduling used in this architecture is not a good fit for RCP as much of the operational parallelism is subsumed by the data parallelism of the SIMD execution model. Since each SIMD region can only do a single operation, the algorithm will find the highest priority operation, then schedule that all of the operations of that type into the that region (in priority order). This undercuts the utility of the distance metric and the slack metric.

Most of the variation is seen in SHA-1 $k = 2$, where prioritizing for distance actually reduces the effectiveness of the algorithm, but slack improves it slightly.

Unfortunately, this seems to be a limited case, with the difference disappearing by $k = 4$.

### 4.2.3 LPFS Configuration Variability

Figure 4.6 shows the speedups of LPFS at $k = 2$ and 4 with different parameter configurations. LPFS has three configurable parameters: $l$ is the number of longest paths to schedule, SIMD controls whether SIMD operation is used, and Refill which allows completed paths to have a new path scheduled in the newly freed SIMD region. The Refill option (only usable with SIMD) has no appreciable effect on the speedup. Reusing a SIMD region after completing a path is rarely needed; most modules don't have many second and third longest paths that are significantly shorter than the longest path and the new longest path may introduce stalls because dependencies aren't met yet. SIMD typically has a moderate impact on the overall runtime, the only exception being in GSE, where non-SIMD can be better. This is likely due to the register-based structure which performs longer sequences of operations on the same qubits, reducing the amount of communication incurred significantly when SIMD is not used. There may also be some artifacts in the free-list handling of LPFS that assist with GSE schedules.

TFP's outlier at $k = 4$, $l = 1$ is predominantly due to the flexible boundary scheduling. Since $l + 1$ is the minimum size of any LPFS scheduled module,

flexible boundaries only work at this node, because all leaves will require 2-4 SIMD regions. At $l > 1$, the widths increase to a minimum of 3, so no coarse-grained parallelism can be attained at $k = 4$. Revising LPFS to allow for a minimum width based on actual usage, instead of $l + 1$ should help with coarse-grained scheduling.

## 4.3 Communication Costs, Requirements and Limits

Communication is interesting in quantum computing because the actual transmission of the quantum state is functionally instantaneous (at least 10,000 times faster than the speed of light [66]), so the actual bandwidth is limited solely by the overhead of preparing the EPR pairs, transmitting one half of the pair to the destination, transmission of classical state information, and performing the measurement and reconstruction operations. This allows a functional maximum bandwidth for the system of however many measurement operations (less than or equal to $k$ times $d$ times 2, assuming full-duplex) can be performed in a single timestep. The overhead costs are expensive, though, requiring four timesteps for each movement phase and physical bandwidth to transmit the EPR pairs between their source and destination SIMD regions and memory.

In order to investigate further realism in the simulation of the Multi-SIMD architecture, the system-wide communication costs were measured as well. The movements within the system were tracked on a qubit and a cycle-by-cycle basis in order to determine the sustained throughput and peak bandwidth of the system. These metrics allow for reasoning about the system-level needs for provisioning inter-region EPR bandwidth and teleportation overhead calculations.

### 4.3.1 Sustained Throughput Requirements

Sustained throughput shown in Figure 4.7 is the average number of qubits moved in a single cycle over the entire runtime of the benchmark, including timesteps involved in performing movements (since throughput is defined as including overhead costs). SHA-1 has the highest requirement of an average of around 2.6 qubits moved at every timestep, due to the high amount of data parallelism in the algorithm as well as the nature of block-base hashing algorithms. Most other algorithms average less than one qubit per cycle, though this is realistically higher when only considering computation timesteps that are not involved in performing movement. GSE is particularly low, but this is explained by the large speedup seen between Figures 4.1 and 4.3 when movement was considered. These results give a minimum value for an EPR pair generation rate that must sustained throughout the execution of the benchmark. Dropping below this level will

cause throttling of the system to deal with insufficient communication, resulting in longer runtimes.

Though communication shows an increase with $k$, this is expected as more resources are available and does not appreciably increase for additional SIMD regions.

### 4.3.2 Peak Bandwidth Limits

The peak bandwidth in Figure 4.8 is plotted against a log scale due to the wide variation of peaks between algorithms. Most benchmarks show a peak level that doesn't vary between algorithms or configurations; this is largely due to the nature of both the hierarchical scheduling model and the maximum data width of the benchmark, typically dictated by the problem size. To allow for hierarchical blackbox scheduling, the compiler assumes that all of the active qubits will be flushed to the global memory and the module will start retrieving all of the qubits it needs directly from memory. This approach is obviously inefficient and will be corrected in future versions of the compiler. The maximum data width of the benchmark refers to the typically width passed by either a critical module or by most of the modules and typically related to the problem size. Grover's algorithm is dominated by *initialize* and *measure* modules which act on all non-ancilla qubits in the algorithm, even though most modules only pass

a few qubits at a time; pipelining or serializing Grover's in these modules can dramatically reduce the peak bandwidth load. GSE's register based approach has the contents of two registers passed to almost every module; in this case, moves may be further eliminated as the active qubits were the same in both contexts. SHA-1 shows increases in each algorithm, showing that as the amount of available data parallelism is increased, the algorithm can be scaled with it, implying that the benchmark has data parallelism to spare, even with up to 4096 qubits worth of processing (with $k = 4$, $d = 1024$). SHA-1 will only continue to increase as $k$ does.

Currently the architecture assumes an infinite capacity to generate and store EPRs, but the communication costs shown here definitely will force realistic solutions to the outrageous number of EPR pairs that need to be managed. Strategies may include reusing disentangled EPR qubits, buffering, or even throttling execution to allow for underruns in the EPR pair resources.

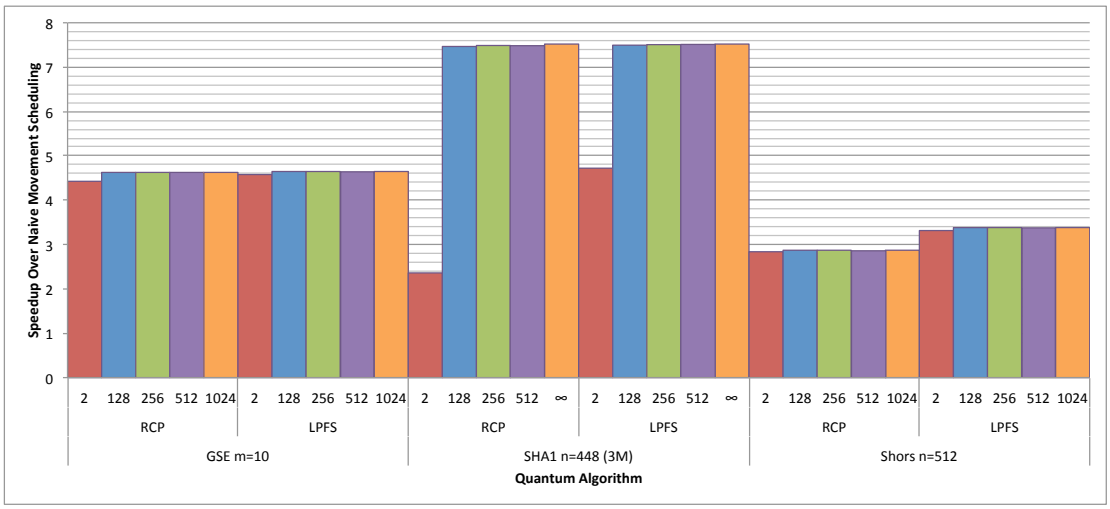


Figure 4.4: The speedup of GSE, SHA-1, and Shor's algorithm with respect to $d$, including communication.

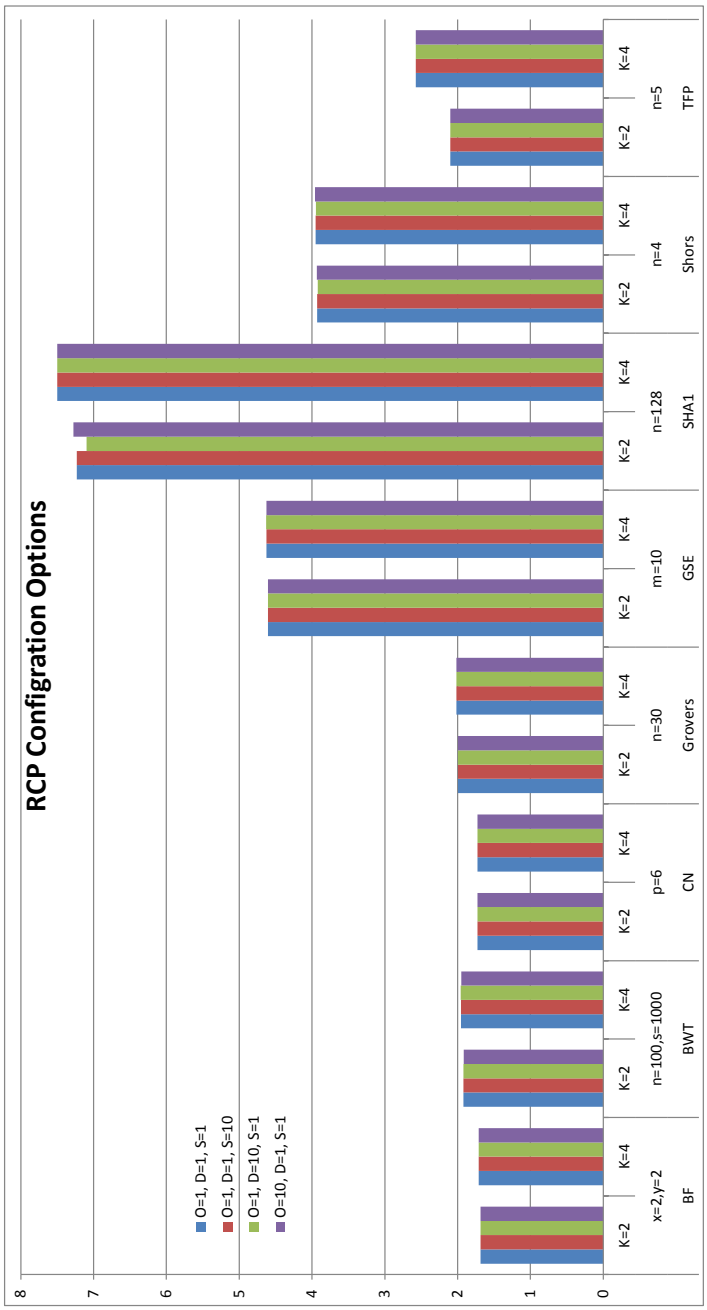


Figure 4.5: Speedups with movement costs, varying the RCP options. The weights for scheduling priority are based on Operation type (O), Distance (D), and Slack (S). Little variation is seen.

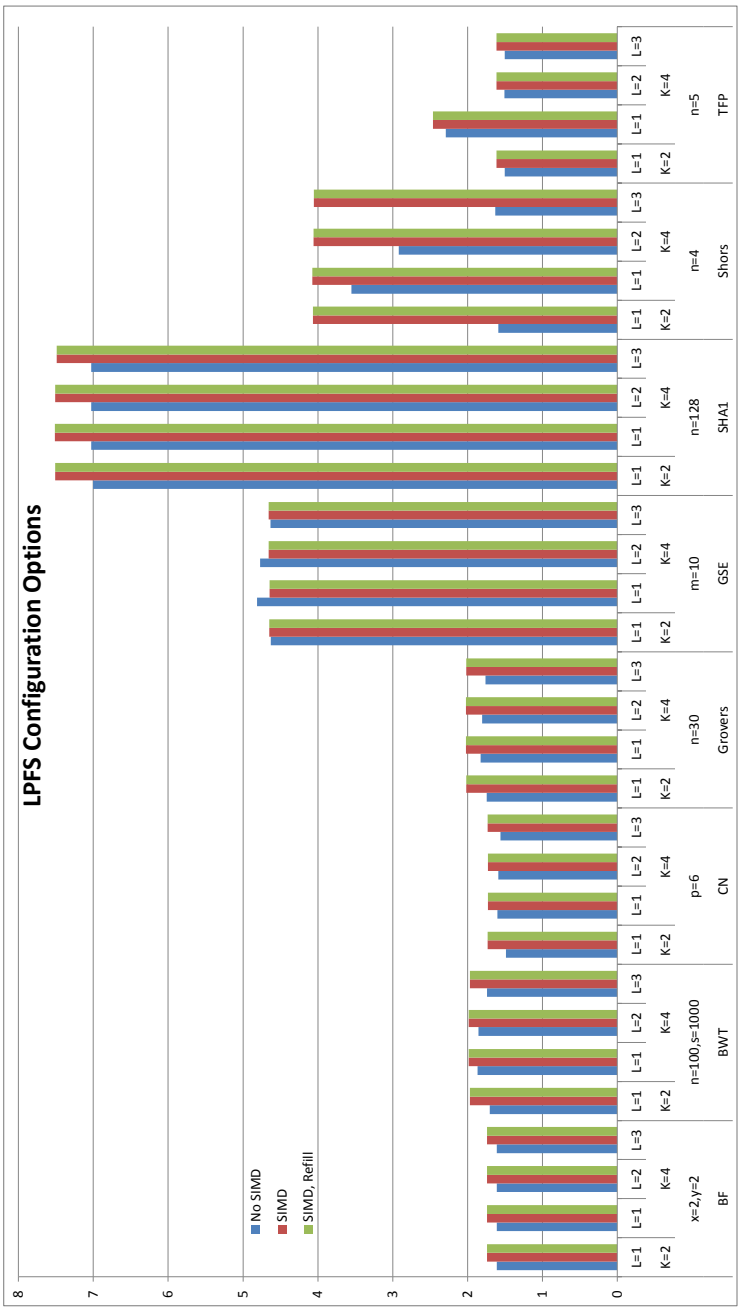


Figure 4.6: Speedups with movement costs, varying the LPFS options. Options include $l$, SIMD and refill. Typically $l = 1$ is preferred, as is SIMD operation, though GSE has an odd outlier for non-SIMD.

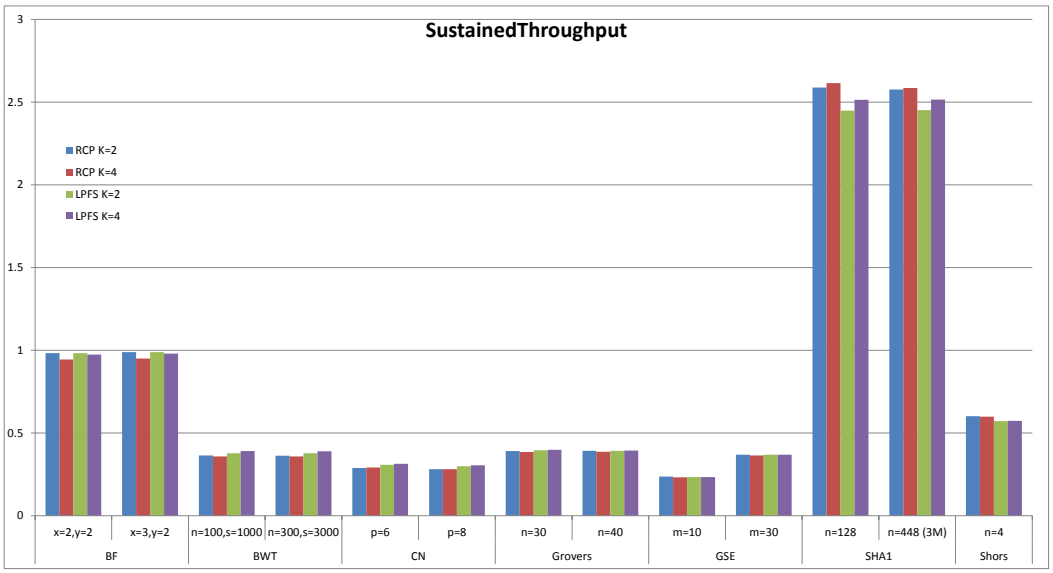


Figure 4.7: The average throughput of the benchmark over its execution, in qubits per cycle. Teleportation overhead cycles are included in the total runtime. This gives a minimum communication cost need to sustain operation.

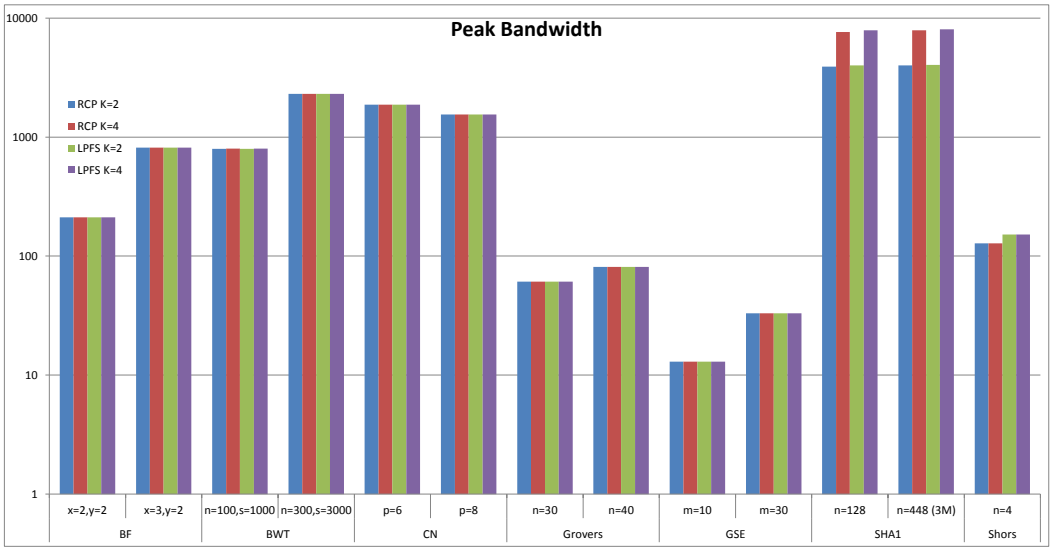


Figure 4.8: The peak bandwidth (or most moves seen in a single cycle) of a benchmark. This gives an upper bound for ideal scheduling of benchmarks.

# Chapter 5

# Related Work

This work significantly based on work in [22, 26]. Some of the work is present both here and in these papers. The following statements are taken from [22].

This work builds on several important previous studies relating to SIMD parallelism [8, 29, 49], ancilla preparation [24, 28], and quantum architecture [8, 39, 58]. This work is the second to use a complete compiler infrastructure to discover this parallelism, allowing evaluation of a non-trivial set of benchmarks (previous work focused almost exclusively on Shor's and Grover's algorithm, or other small quantum circuits). It is also the second to incorporate data movement analysis and optimizations within the compiler framework established. Additionally, it focuses more completely on the scheduling algorithms used to determine runtime.

Parallel work has been done by the Quipper team [18]. They have developed a very similar system based on Haskell which explores compiliation, circuit generation, and execution of quantum circuits. They were a part of the IARPA

funded research that started this project at UCSB. Their work was conducted independently of this work, but I have been present for some of their presentations and have discussed technical details of one of the algorithms (Quantum Linear Systems, non-functional in this work) with them.

Some prior work has explored optimization of execution latencies with SIMD architectures, but in a more limited context. Chi *et.al.* [8] proposed a SIMD architecture based on the technology of *electron spins on liquid helium*. For a quantum carry-lookahead adder circuit, they evaluated pipelining of ancilla preparation for CNOT and Toffoli gates to reduce latency, and optimization of width of SIMD regions to reduce area requirements. This work builds on this model, with the implementation of a complete compiler and the study of a much larger and more diverse benchmark suite.

Schuchman *et.al.* [49] identify a high-level parallelism pertaining to specific quantum tasks of *uncomputation* (analogous to garbage collection for qubits) and propose a multi-core architecture to minimize latency and expensive inter-core communication during their execution. This kind of parallelism fits well into the Multi-SIMD model; it can be easily extended to support the proposed multiple cores. Some degree of uncomputation already exists in the compiled code of the benchmarks and is naturally parallelized by the model, and more can be added in the future to reclaim unused qubits.

The SIMD regions in the architecture are well-suited for a commonly used class of error-correction codes known as concatenated codes [2, 53]. A new class of ensemble codes, known as surface codes [23], have the potential of lowering ECC overhead for very large problems. Future research will explore whether surface code operations are amenable to SIMD parallelism.

# Chapter 6

# Future Work

There is much more work to be done in this vein of research. The first steps are merging LPFS into the ScaffCC/LLVM framework. By doing this, it can be promoted to a full program scheduler, not merely for leaf nodes. Additionally, LPFS has some limitations in the prioritization of its free list, which can be improved. Enabling LPFS to perform some hierarchical scheduling as well will improve scheduling in non-leaf modules, which currently use inefficient blackboxing. These changes should impact both the actual schedules produced as well as the overall runtime and memory of the scheduling algorithms.

The algorithms themselves may be re-written in more parallel-friendly ways, either by introducing redundant computation that can be merged probabilistically or by restructuring the algorithm to decrease data dependencies.

This work draws heavily from [22], which also discusses a local memory optimization that can be used to reduce teleportation costs. By continuing to push these analyses into ScaffCC, the toolchain becomes more robust and useful.

Future analysis work includes refining tracking of communication costs by accurately modeling qubit movements at module boundaries. While this is likely a minimal change to throughput, it may reduce the peak bandwidth usage to something that is more practically achievable. Additional experimentation can be done by scaling $d$ to limit available bandwidth based on modeling of EPR generation and distribution, as well as looking at multi-hop (all communication through global memory) instead of fully-connected, single-hop transmission as explored here.

Finally, incorporating QECC costs for both qubits and computation and ancilla preparation will allow for complete end-to-end scheduling of the benchmark algorithms.

# Chapter 7

# Conclusion

The Multi-SIMD architecture proposed here allows for reasoning about the practical physical constraints of a quantum computer. By using the ScaffCC toolchain to build gate-level programs that are highly scalable, the computational time as well as the data communication within the system can be analyzed. Computation is shown to be dominated by the 80% communication overhead. By limiting the movement within the system coupled with the parallelism in the architecture, speedups of between 1.6X and 7.9X over naive movement models are seen. These results are based on logical-level operations, and the incorporation of quantum error correction (QECC) can result in exponential overhead costs [2, 44, 53]. By providing even limited speedup, the computations may be able to avoid using increasing levels of QECC and significantly reduce the runtime further.

The architectural costs of generating up to 2.5 qubits per cycle on average and managing qubit buffers of $2 * kd$ EPR pairs for communication will require a large technical effort to achieve. More satisifactory architectural trade-offs will need to be found in order to create a more viable quantum computer model.

# Bibliography


[1] D-wave systems, March 2014.

[2] Panos Aliferis and Andrew Cross. Subsystem fault tolerance with the bacon-shor code. *arXiv preprint quant-ph/0610063*, 2006.

[3] D. T. C. Allcock, T. P. Harty, C. J. Ballance, B. C. Keitch, N. M. Linke, D. N. Stacey, and D. M. Lucas. A microfabricated ion trap with integrated microwave circuitry. *Applied Physics Letters*, 102(4):–, 2013.

[4] Andris Ambainis, Andrew M. Childs, Ben W. Reichardt, Robert Spalek, and Shengyu Zhang. Any AND-OR Formula of Size N Can Be Evaluated in Time $N^{1/2+O(1)}$ on a Quantum Computer. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '07, pages 363–372, Washington, DC, USA, 2007. IEEE Computer Society.

[5] John Bell. *Speakable and unspeakable in quantum mechanics.* The Press Syndicate of the University of Cambridge, 1987.

[6] BB Blinov, DL Moehring, L-M Duan, and Chris Monroe. Observation of entanglement between a single trapped atom and a single photon. *Nature*, 428(6979):153–157, 2004.

[7] K. R. Brown, A. C. Wilson, Y. Colombe, C. Ospelkaus, A. M. Meier, E. Knill, D. Leibfried, and D. J. Wineland. Single-qubit-gate error below $\mathbf{10^{-4}}$ in a trapped ion. *Phys. Rev. A*, 84:030303, Sep 2011.

[8] Eric Chi, Stephen A. Lyon, and Margaret Martonosi. Tailoring quantum architectures to implementation style: a quantum computer for mobile and persistent qubits. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 198–209, New York, NY, USA, 2007. ACM.

[9] J Chiaverini, D Leibfried, T Schaetz, MD Barrett, RB Blakestad, J Britton, WM Itano, JD Jost, E Knill, C Langer, R Ozeri, and D. J. Wineland. Realization of quantum error correction. *Nature*, 432(7017):602–605, 2004.

[10] Andrew M. Childs, Richard Cleve, Enrico Deotto, Edward Farhi, Sam Gutmann, and Daniel A. Spielman. Exponential algorithmic speedup by a quantum walk. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, STOC '03, pages 59–68, New York, NY, USA, 2003. ACM.

[11] Diana P. L. Aude Craik, N. M. Linke, T. P. Harty, C. J. Ballance, D. M. Lucas, A. M. Steane, and D. T. C. Allcock. Microwave control electrodes for scalable, parallel, single-qubit operations in a surface-electrode ion trap, August 2013.

[12] Steven A Cuccaro et al. A New Quantum Ripple-Carry Addition Circuit. *arXiv preprint quant-ph/0410184*, 2004.

[13] Christopher M. Dawson and Michael A. Nielsen. The Solovay-Kitaev algorithm. *Quantum Info. Comput.*, 2006.

[14] David P DiVincenzo. The physical implementation of quantum computation. *arXiv preprint quant-ph/0002077*, 2000.

[15] Artur Ekert and Richard Jozsa. Quantum computation and shor's factoring algorithm. *Rev. Mod. Phys.*, 68:733–753, Jul 1996.

[16] JJ Garcıa-Ripoll, P Zoller, and JI Cirac. Speed optimized two-qubit gates with laser coherent control techniques for ion trap quantum computing. *Phys. Rev. Lett*, 91(15):157901, 2003.

[17] Mark Gebhart, Bertrand A. Maher, Katherine E. Coons, Jeff Diamond, Paul Gratz, Mario Marino, Nitya Ranganathan, Behnam Robatmili, Aaron Smith, James Burrill, Stephen W. Keckler, Doug Burger, and Kathryn S. McKinley.

An evaluation of the TRIPS computer system. In *In Proceedings of the Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, March 2009.

[18] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. *SIGPLAN Not.*, 48(6):333–342, June 2013.

[19] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, STOC '96, pages 212–219, New York, NY, USA, 1996. ACM.

[20] Sean Hallgren. Fast Quantum Algorithms for Computing the Unit Group and Class Group of a Number Field. In *Symposium on Theory of Computing*. ACM, 2005.

[21] Aram W Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum Algorithm for Linear Systems of Equations. *Physical Review Letters*, 103(15):150502, 2009.

[22] Jeff Heckey, Shruti Patil, Ali JavadiAbhari, Adam Holmes, Daniel Kudrow, Frederic T. Chong, Margaret Martonosi, Ken Brown, and Diana Franklin. [submitted] compiler management of communication and parallelism for quantum computation. *ASPLOS 2015*, 2014.

[23] Clare Horsman, Austin G Fowler, Simon Devitt, and Rodney Van Meter. Surface code quantum computing by lattice surgery. *New Journal of Physics*, 14(12):123011, 2012.

[24] Nemanja Isailovic, Mark Whitney, Yatish Patel, and John Kubiatowicz. Running a quantum circuit at the speed of data. In *ACM SIGARCH Computer Architecture News*. IEEE Computer Society, 2008.

[25] Ali JavadiAbhari et al. Scaffold: Quantum Programming Language. Technical report, Princeton University, NJ, USA, 2012.

[26] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T Chong, and Margaret Martonosi. Scaffcc: A Framework for Compilation and Analysis of Quantum Computing Programs. *ACM International Conference on Computing Frontiers (CF 2014)*, 2014.

[27] M. Johanning, A. Braun, N. Timoney, V. Elman, W. Neuhauser, and Chr. Wunderlich. Individual addressing of trapped ions and coupling of motional and spin states using rf radiation. *Phys. Rev. Lett.*, 102:073004, Feb 2009.

[28] N Cody Jones, Rodney Van Meter, Austin G Fowler, Peter L McMahon, Jungsang Kim, Thaddeus D Ladd, and Yoshihisa Yamamoto. Layered architecture for quantum computing. *Physical Review X*, 2(3):031007, 2012.

[29] J Kim, S Pau, Z Ma, HR McLellan, JV Gates, A Kornblit, Richard E Slusher, Robert M Jopson, I Kang, and M Dinu. System design for large-scale ion trap quantum information processor. *Quantum Information & Computation*, 5(7):515–537, 2005.

[30] Vadym Kliuchnikov, Dmitri Maslov, and Michele Mosca. Practical Approximation of Single-Qubit Unitaries by Single-Qubit Quantum Clifford and T Circuits. *arXiv preprint arXiv:1212.6964*, 2012.

[31] Vadym Kliuchnikov, Dmitri Maslov, and Michele Mosca. Asymptotically Optimal Approximation of Single Qubit Unitaries by Clifford and T Circuits Using a Constant Number of Ancillary Qubits. *Physical review letters*, 110(19):190502, 2013.

[32] Vadym Kliuchnikov, Dmitri Maslov, and Michele Mosca. Fast and Efficient Exact Synthesis of Single-Qubit Unitaries Generated by Clifford and T Gates. *Quantum Information & Computation*, 13(7-8):607–630, 2013.

[33] Daniel Kudrow et al. Quantum Rotations: A Case Study in Static and Dynamic Machine-Code Generation for Quantum Computers. In *International Symposium on Computer Architecture*. ACM, 2013.

[34] Leo Lapworth. Perl profiling with devel::nytprof (the perl profiler).

[35] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization*, 2004.

[36] Dietrich Leibfried, Brian DeMarco, Volker Meyer, David Lucas, Murray Barrett, Joe Britton, B Jelenkovi&cacute WM Itano, C Langer, and DJ T Rosenband. Experimental demonstration of a robust, high-fidelity geometric two ion-qubit phase gate. *Nature*, 422(6930):412–415, 2003.

[37] Daniel Loss and David P DiVincenzo. Quantum computation with quantum dots. *Physical Review A*, 57(1):120, 1998.

[38] Frédéric Magniez, Miklos Santha, and Mario Szegedy. Quantum algorithms for the triangle problem. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '05, pages 1109–1117, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[39] Tzvetan S. Metodi, Darshan D. Thaker, and Andrew W. Cross. A quantum logic array microarchitecture: Scalable quantum data movement and computation. In *MICRO*, pages 305–318. IEEE Computer Society, 2005.

[40] Chris Monroe, DM Meekhof, BE King, WM Itano, and DJ Wineland. Demonstration of a fundamental quantum logic gate. *Physical Review Letters*, 75(25):4714, 1995.

[41] Michele Mosca. Quantum algorithms. In Robert A. Meyers, editor, *Encyclopedia of Complexity and Systems Science*, pages 7088–7118. Springer New York, 2009.

[42] Michael A Nielsen and Isaac L Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2010.

[43] National Institute of Standards and Technology. *FIPS PUB 180-4: Secure Hash Standard (SHS)*. U.S. Department of Commerce, 2012.

[44] M. Oskin, F.T. Chong, and I.L. Chuang. A Practical Architecture for Reliable Quantum Computers. *Computer*, 35(1):79–87, 2002.

[45] C. Ospelkaus, U. Warring, Y. Colombe, K. R. Brown, J. M. Amini, D. Leibfried, and D. J. Wineland. Microwave quantum logic gates for trapped ions. *Nature*, 476:181–184, 2011.

[46] Archimedes Pavlidis and Dimitris Gizopoulos. Fast Quantum Modular Exponentiation Architecture for Shor's Factoring Algorithm. *Quantum Information and Computation*, 14:0649–0682, 2014.

[47] Mark Riebe, H Häffner, CF Roos, W Hänsel, J Benhelm, GPT Lancaster, TW Körber, C Becher, F Schmidt-Kaler, and DFV James. Deterministic quantum teleportation with atoms. *Nature*, 429(6993):734–737, 2004.

[48] Ferdinand Schmidt-Kaler, Hartmut Häffner, Mark Riebe, Stephan Gulde, Gavin PT Lancaster, Thomas Deuschle, Christoph Becher, Christian F Roos, Jürgen Eschner, and Rainer Blatt. Realization of the cirac–zoller controlled-not quantum gate. *Nature*, 422(6930):408–411, 2003.

[49] Ethan Schuchman and T. N. Vijaykumar. A Program Transformation and Architecture Support for Quantum Uncomputation. In *Architectural Support for Programming Languages and Operating Systems*. ACM, 2006.

[50] C. M. Shappert, J. T. Merrill, K. R. Brown, J. M. Amini, C. Volin, S. C. Doret, H. Hayden, C-S. Pai, and A. W. Harter. Spatially uniform single-qubit gate operations with near-field microwaves and composite pulse compensation. *New Journal of Physics*, 15(083053), 2013.

[51] Peter W Shor. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 124–134. IEEE, 1994.

[52] Sqct: Single Qubit Circuit Toolkit - https://code.google.com/p/sqct/, May 2014.

[53] A. Steane. Error correcting codes in quantum theory. *Physical Review Letters*, 77(5):793–797, 1996.

[54] A. M. Steane. Active stabilization, quantum computation, and quantum state synthesis. *Phys. Rev. Lett.*, 78:2252–2255, Mar 1997.

[55] Andrew Steane. Multiple-Particle Interference and Quantum Error Correction. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 452(1954):2551–2577, 1996.

[56] Steven Swanson, Andrew Schwerin, Martha Mercaldi, Andrew Petersen, Andrew Putnam, Ken Michelson, Mark Oskin, and Susan J. Eggers. The WaveScalar architecture. *ACM Trans. Comput. Syst.*, 25(2):4:1–4:54, May 2007.

[57] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 2–. IEEE Computer Society, 2004.

[58] Darshan D. Thaker, Tzvetan S. Metodi, Andrew W. Cross, Isaac L. Chuang, and Frederic T. Chong. Quantum memory hierarchies: Efficient designs to

match available parallelism in quantum computing. In *ISCA*, pages 378–390. IEEE Computer Society, 2006.

[59] U. Warring, C. Ospelkaus, Y. Colombe, K. R. Brown, J. M. Amini, M. Carsjens, D. Leibfried, and D. J. Wineland. Techniques for microwave near-field quantum control of trapped ions. *Phys. Rev. A*, 87:013437, Jan 2013.

[60] James D. Whitfield, Jacob Biamonte, and Alan Aspuru-Guzik. Simulation of electronic structure hamiltonians using quantum computers. *Molecular Physics*, 109(5):735, 2010.

[61] Mark G. Whitney, Nemanja Isailovic, Yatish Patel, and John Kubiatowicz. A fault tolerant, area efficient architecture for shor's factoring algorithm. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 383–394, New York, NY, USA, 2009. ACM.

[62] David J Wineland, C Monroe, WM Itano, D Leibfried, BE King, and DM Meekhof. Experimental issues in coherent quantum-state manipulation of trapped atomic ions. *arXiv preprint quant-ph/9710025*, 1997.

[63] DJ Wineland, C Monroe, WM Itano, BE King, D Leibfried, DM Meekhof, C Myatt, and C Wood. Experimental primer on the trapped ion quantum computer. *spectroscopy*, 7:8, 1998.

[64] Tao Yang and A Gerasoulis. PYRROS: static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the 6th international conference ...*, pages 428–437, 1992.

[65] Tao Yang and Apostolos Gerasoulis. List scheduling with and without communication delays. *Parallel Comput.*, 19(12):1321–1344, December 1993.

[66] Juan Yin, Yuan Cao, Hai-Lin Yong, Ji-Gang Ren, Hao Liang, Sheng-Kai Liao, Fei Zhou, Chang Liu, Yu-Ping Wu, Ge-Sheng Pan, Li Li, Nai-Le Liu, Qiang Zhang, Cheng-Zhi Peng, and Jian-Wei Pan. Lower bound on the speed of nonlocal correlations without locality and measurement choice loopholes. *Phys. Rev. Lett.*, 110:260407, Jun 2013.

# Appendix A

# Supplementary Data

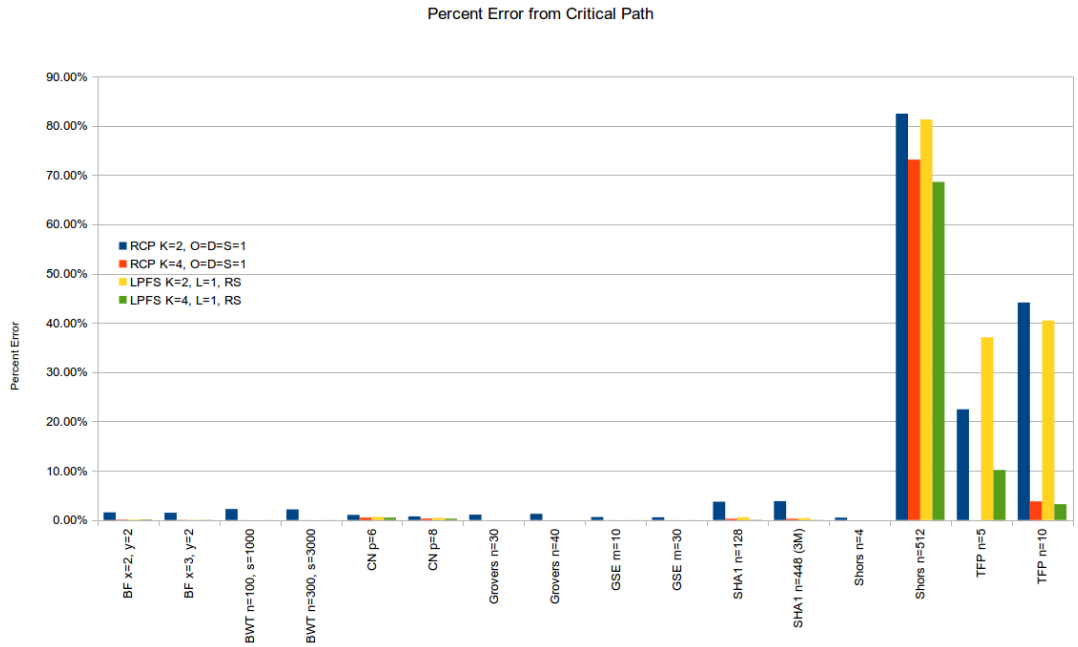


Figure A.1: The percent error of the scheduling algorithms from the ideal speedup. Largely dominated by the low $k$ values for Shor's, though TFP can also be improved with higher $k$.

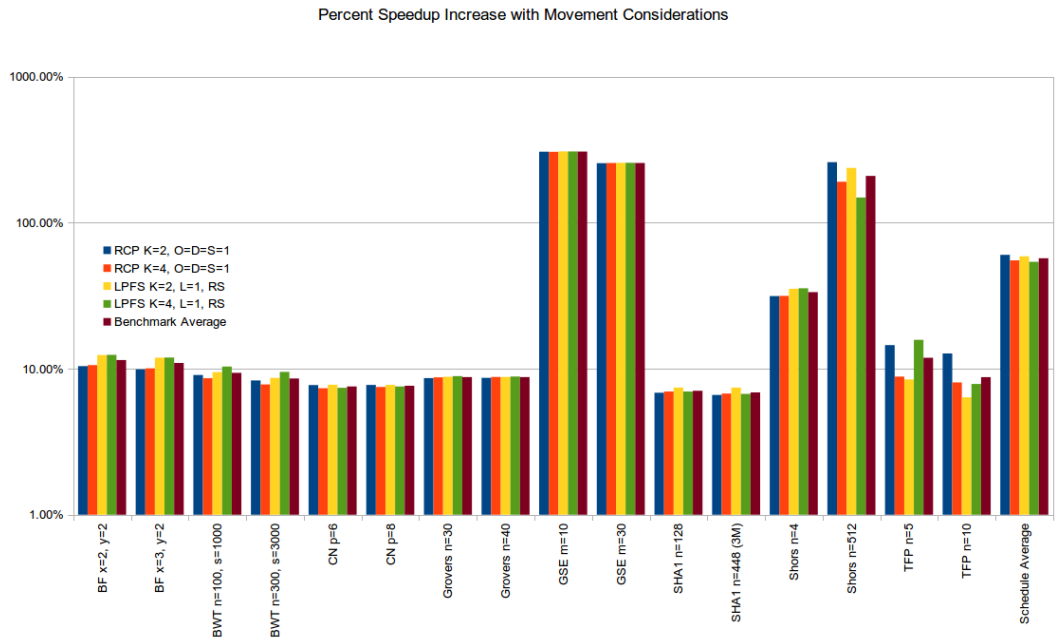


Figure A.2: The percentage of increase in speedup between runtimes without movement costs and with movement costs, plotted logarithmically. All speedups improve, typically nearly constant values within a scheduling algorithm. In more serial algorithms, LPFS shows greater reduction in total moves (larger gains). Both Shor's and TFP seem to benefit more from increased instruction level parallelism (higher $k$) than from reduced data overhead.

# Appendix B

# Example Schedule

The following is a small example of the output produced while scheduling the file *qft.scaffold*. Each section describes the output generated. Only a single schedule configuration is followed: LPFS scheduling with $k = 4$, $d = 1024$, $l = 1$, SIMD and Refill enabled.

## B.1 Source Code

The source of *qft.scaffold*.

```
#include <math.h>

#define pi 3.14159265358979323846264338327950288419 7

module PhasePi8 ( qbit bit ) {
    /* PhasePi8 matrix:
       [ [ 1 0     ]
         [ 0 e^i*pi/8 ] ]

    Notes: Can be decomposed as Rz(-pi/8)*[ [e^i*pi/16 0]
                                            [0 e^i*pi/16] ]

     */

     Rz(bit, -1*pi/8);
}

module cT ( qbit ctrl, qbit target ) {
    /* cT identity matrix:
       [ [ 1 0 0      0     ]
         [ 0 1 0      0     ]
         [ 0 0 1      0     ]
         [ 0 0 0 e^i*pi/4 ] ]
    */

    PhasePi8(ctrl);
```

```
    Rz(target,pi/8);
    CNOT(target,ctrl);
    Rz(target,-1*pi/8);
    CNOT(target,ctrl);

}

module cS ( qbit ctrl, qbit target ) {
    /* cS identity matrix:
       [ [ 1 0 0 0 ]
         [ 0 1 0 0 ]
         [ 0 0 1 0 ]
         [ 0 0 0 i ] ]
    */

    T(ctrl);
    Rz(target,pi/4);
    CNOT(target,ctrl);
    Rz(target,-1*pi/4);
    CNOT(target,ctrl);

}

module cRz ( qbit ctrl, qbit target, const double angle ) {
    /* cRz identity matrix:
       [ [ 1 0 0                 0     ]
         [ 0 1 0                 0     ]
         [ 0 0 e^(-i*angle/2)    0     ]
         [ 0 0 0                 e^(i*angle/2) ] ]
    */

    Rz(target,-1*angle/2);
    CNOT(target,ctrl);
    Rz(target,angle/2);
    CNOT(target,ctrl);

}


module qft5 ( qbit bit[5] ) {
    H   ( bit[0] );
    cS  ( bit[0], bit[1] );
    H   ( bit[1] );
    cT  ( bit[0], bit[2] );
    cS  ( bit[1], bit[2] );
    H   ( bit[2] );
    cRz ( bit[0], bit[3], pi/8 );
    cT  ( bit[1], bit[3] );
    cS  ( bit[2], bit[3] );
    H   ( bit[3] );
    cRz ( bit[0], bit[4], pi/16 );
    cRz ( bit[1], bit[4], pi/8 );
    cT  ( bit[2], bit[4] );
    cS  ( bit[3], bit[4] );
    H   ( bit[4] );
}

int main () {
    qbit reg[5];
    cbit out[5];
```

```
    int i, qft;

    for ( i = 0; i < 5; i++ ) {
        PrepZ( reg[i],0 );
    }
    qft5( reg );
    for ( i = 0; i < 5; i++ ) {
        out[i] = MeasX( reg[i] );
    }
    return 0;
}
```

## B.2 Flattened LLVM Output

After compilation with LLVM, the LLVM IR is generated. All rotations have been decomposed and all loops have been unrolled. This code is then analyzed and flattened to a maximum of 2 million gates in a single module.

Due to the overall length the rotation decompositions are excerpted.

```
; ModuleID = 'qft.ll'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-↩
    f32:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8↩
    :16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

declare void @llvm.CNOT(i16, i16) nounwind

declare void @llvm.T(i16) nounwind

declare void @llvm.H(i16) nounwind

define i32 @main() nounwind {
entry:
  %reg = alloca [5 x i16], align 2
  %out = alloca [5 x i1], align 1
  %arrayidx = getelementptr inbounds [5 x i16]* %reg, i64 0, i64 0
  %0 = load i16* %arrayidx, align 2
  call void @llvm.PrepZ(i16 %0, i32 0)
  %arrayidx.1 = getelementptr inbounds [5 x i16]* %reg, i64 0, i64 1
  %1 = load i16* %arrayidx.1, align 2
  call void @llvm.PrepZ(i16 %1, i32 0)
  %arrayidx.2 = getelementptr inbounds [5 x i16]* %reg, i64 0, i64 2
  %2 = load i16* %arrayidx.2, align 2
  call void @llvm.PrepZ(i16 %2, i32 0)
  %arrayidx.3 = getelementptr inbounds [5 x i16]* %reg, i64 0, i64 3
  %3 = load i16* %arrayidx.3, align 2
  call void @llvm.PrepZ(i16 %3, i32 0)
  %arrayidx.4 = getelementptr inbounds [5 x i16]* %reg, i64 0, i64 4
  %4 = load i16* %arrayidx.4, align 2
  call void @llvm.PrepZ(i16 %4, i32 0)
  %arraydecay = getelementptr inbounds [5 x i16]* %reg, i64 0, i64 0
  %5 = load i16* %arraydecay, align 2
  call void @llvm.H(i16 %5) nounwind
  %6 = load i16* %arraydecay, align 2
  %arrayidx2.i = getelementptr inbounds i16* %arraydecay, i64 1
```

```
%7 = load i16* %arrayidx2.i, align 2
call void @llvm.T(i16 %6) nounwind
call void @llvm.Tdag(i16 %7) nounwind
call void @llvm.Tdag(i16 %7) nounwind
...
call void @llvm.H(i16 %7) nounwind
call void @llvm.S(i16 %7) nounwind
call void @llvm.CNOT(i16 %7, i16 %6) nounwind
%arrayidx3.i = getelementptr inbounds i16* %arraydecay, i64 1
%8 = load i16* %arrayidx3.i, align 2
call void @llvm.H(i16 %8) nounwind
%9 = load i16* %arraydecay, align 2
%arrayidx5.i = getelementptr inbounds i16* %arraydecay, i64 2
%10 = load i16* %arrayidx5.i, align 2
call void @llvm.Tdag(i16 %9) nounwind
call void @llvm.Tdag(i16 %9) nounwind
call void @llvm.Tdag(i16 %9) nounwind
...
call void @llvm.X(i16 %10) nounwind
call void @llvm.Sdag(i16 %10) nounwind
call void @llvm.CNOT(i16 %10, i16 %9) nounwind
%arrayidx6.i = getelementptr inbounds i16* %arraydecay, i64 1
%11 = load i16* %arrayidx6.i, align 2
%arrayidx7.i = getelementptr inbounds i16* %arraydecay, i64 2
%12 = load i16* %arrayidx7.i, align 2
call void @llvm.T(i16 %11) nounwind
call void @llvm.Tdag(i16 %12) nounwind
call void @llvm.Tdag(i16 %12) nounwind
...
call void @llvm.Z(i16 %12) nounwind
call void @llvm.H(i16 %12) nounwind
call void @llvm.S(i16 %12) nounwind
call void @llvm.CNOT(i16 %12, i16 %11) nounwind
%arrayidx8.i = getelementptr inbounds i16* %arraydecay, i64 2
%13 = load i16* %arrayidx8.i, align 2
call void @llvm.H(i16 %13) nounwind
%14 = load i16* %arraydecay, align 2
%arrayidx10.i = getelementptr inbounds i16* %arraydecay, i64 3
%15 = load i16* %arrayidx10.i, align 2
call void @llvm.Tdag(i16 %15) nounwind
call void @llvm.Tdag(i16 %15) nounwind
call void @llvm.Tdag(i16 %15) nounwind
...
call void @llvm.H(i16 %15) nounwind
call void @llvm.T(i16 %15) nounwind
call void @llvm.X(i16 %15) nounwind
call void @llvm.CNOT(i16 %15, i16 %14) nounwind
%arrayidx11.i = getelementptr inbounds i16* %arraydecay, i64 1
%16 = load i16* %arrayidx11.i, align 2
%arrayidx12.i = getelementptr inbounds i16* %arraydecay, i64 3
%17 = load i16* %arrayidx12.i, align 2
call void @llvm.Tdag(i16 %16) nounwind
call void @llvm.Tdag(i16 %16) nounwind
call void @llvm.Tdag(i16 %16) nounwind
...
call void @llvm.T(i16 %17) nounwind
call void @llvm.X(i16 %17) nounwind
call void @llvm.Sdag(i16 %17) nounwind
call void @llvm.CNOT(i16 %17, i16 %16) nounwind
%arrayidx13.i = getelementptr inbounds i16* %arraydecay, i64 2
```

```
%18 = load i16* %arrayidx13.i, align 2
%arrayidx14.i = getelementptr inbounds i16* %arraydecay, i64 3
%19 = load i16* %arrayidx14.i, align 2
call void @llvm.T(i16 %18) nounwind
call void @llvm.Tdag(i16 %19) nounwind
call void @llvm.Tdag(i16 %19) nounwind
...
call void @llvm.H(i16 %19) nounwind
call void @llvm.S(i16 %19) nounwind
call void @llvm.CNOT(i16 %19, i16 %18) nounwind
%arrayidx15.i = getelementptr inbounds i16* %arraydecay, i64 3
%20 = load i16* %arrayidx15.i, align 2
call void @llvm.H(i16 %20) nounwind
%21 = load i16* %arraydecay, align 2
%arrayidx17.i = getelementptr inbounds i16* %arraydecay, i64 4
%22 = load i16* %arrayidx17.i, align 2
call void @llvm.Tdag(i16 %22) nounwind
call void @llvm.Tdag(i16 %22) nounwind
call void @llvm.Tdag(i16 %22) nounwind
...
call void @llvm.Z(i16 %22) nounwind
call void @llvm.H(i16 %22) nounwind
call void @llvm.Sdag(i16 %22) nounwind
call void @llvm.CNOT(i16 %22, i16 %21) nounwind
%arrayidx18.i = getelementptr inbounds i16* %arraydecay, i64 1
%23 = load i16* %arrayidx18.i, align 2
%arrayidx19.i = getelementptr inbounds i16* %arraydecay, i64 4
%24 = load i16* %arrayidx19.i, align 2
call void @llvm.Tdag(i16 %24) nounwind
call void @llvm.Tdag(i16 %24) nounwind
call void @llvm.Tdag(i16 %24) nounwind
...
call void @llvm.H(i16 %24) nounwind
call void @llvm.T(i16 %24) nounwind
call void @llvm.X(i16 %24) nounwind
call void @llvm.CNOT(i16 %24, i16 %23) nounwind
%arrayidx20.i = getelementptr inbounds i16* %arraydecay, i64 2
%25 = load i16* %arrayidx20.i, align 2
%arrayidx21.i = getelementptr inbounds i16* %arraydecay, i64 4
%26 = load i16* %arrayidx21.i, align 2
call void @llvm.Tdag(i16 %25) nounwind
call void @llvm.Tdag(i16 %25) nounwind
call void @llvm.Tdag(i16 %25) nounwind
...
call void @llvm.T(i16 %26) nounwind
call void @llvm.X(i16 %26) nounwind
call void @llvm.Sdag(i16 %26) nounwind
call void @llvm.CNOT(i16 %26, i16 %25) nounwind
%arrayidx22.i = getelementptr inbounds i16* %arraydecay, i64 3
%27 = load i16* %arrayidx22.i, align 2
%arrayidx23.i = getelementptr inbounds i16* %arraydecay, i64 4
%28 = load i16* %arrayidx23.i, align 2
call void @llvm.T(i16 %27) nounwind
call void @llvm.Tdag(i16 %28) nounwind
call void @llvm.Tdag(i16 %28) nounwind
...
call void @llvm.Z(i16 %28) nounwind
call void @llvm.H(i16 %28) nounwind
call void @llvm.S(i16 %28) nounwind
call void @llvm.CNOT(i16 %28, i16 %27) nounwind
```

```
  %arrayidx24.i = getelementptr inbounds i16* %arraydecay, i64 4
  %29 = load i16* %arrayidx24.i, align 2
  call void @llvm.H(i16 %29) nounwind
  %arrayidx5 = getelementptr inbounds [5 x i16]* %reg, i64 0, i64 0
  %30 = load i16* %arrayidx5, align 2
  %31 = call i1 @llvm.MeasX(i16 %30)
  %arrayidx7 = getelementptr inbounds [5 x i1]* %out, i64 0, i64 0
  call void @store_cbit(i1 %31, i1* %arrayidx7) nounwind
  store i1 %31, i1* %arrayidx7, align 1
  %arrayidx5.1 = getelementptr inbounds [5 x i16]* %reg, i64 0, i64 1
  %32 = load i16* %arrayidx5.1, align 2
  %33 = call i1 @llvm.MeasX(i16 %32)
  %arrayidx7.1 = getelementptr inbounds [5 x i1]* %out, i64 0, i64 1
  call void @store_cbit(i1 %33, i1* %arrayidx7.1) nounwind
  store i1 %33, i1* %arrayidx7.1, align 1
  %arrayidx5.2 = getelementptr inbounds [5 x i16]* %reg, i64 0, i64 2
  %34 = load i16* %arrayidx5.2, align 2
  %35 = call i1 @llvm.MeasX(i16 %34)
  %arrayidx7.2 = getelementptr inbounds [5 x i1]* %out, i64 0, i64 2
  call void @store_cbit(i1 %35, i1* %arrayidx7.2) nounwind
  store i1 %35, i1* %arrayidx7.2, align 1
  %arrayidx5.3 = getelementptr inbounds [5 x i16]* %reg, i64 0, i64 3
  %36 = load i16* %arrayidx5.3, align 2
  %37 = call i1 @llvm.MeasX(i16 %36)
  %arrayidx7.3 = getelementptr inbounds [5 x i1]* %out, i64 0, i64 3
  call void @store_cbit(i1 %37, i1* %arrayidx7.3) nounwind
  store i1 %37, i1* %arrayidx7.3, align 1
  %arrayidx5.4 = getelementptr inbounds [5 x i16]* %reg, i64 0, i64 4
  %38 = load i16* %arrayidx5.4, align 2
  %39 = call i1 @llvm.MeasX(i16 %38)
  %arrayidx7.4 = getelementptr inbounds [5 x i1]* %out, i64 0, i64 4
  call void @store_cbit(i1 %39, i1* %arrayidx7.4) nounwind
  store i1 %39, i1* %arrayidx7.4, align 1
  ret i32 0
}

declare void @llvm.PrepZ(i16, i32) nounwind

declare i1 @llvm.MeasX(i16) nounwind

declare void @store_cbit(i1, i1*)

declare void @llvm.Tdag(i16) nounwind

declare void @llvm.Z(i16) nounwind

declare void @llvm.S(i16) nounwind

declare void @llvm.X(i16) nounwind

declare void @llvm.Sdag(i16) nounwind

declare void @llvm.Y(i16) nounwind
```

## B.3 QASM Output from LLVM

The QASM output is initially scheduled with a simple list scheduler. This gives the numbers at the begining of the lines. This schedule is ignored.

Due to length the following is excerpted to show relevant structure.

```
SIMD_K 4, SIMD_D 1024

#Function main
#Timestep GateName Operand1 Operand2
1 PrepZ reg1
2 Tdag reg1
3 Tdag reg1
...
340 T reg1
341 H reg1
342 T reg1
1 PrepZ reg0
2 H reg0
343 Y reg1
344 H reg1
3 T reg0
345 CNOT reg1 reg0
346 Tdag reg1
...
647 T reg1
1 PrepZ reg2
648 H reg1
2 Tdag reg2
649 Tdag reg1
650 Z reg1
3 Tdag reg2
4 Tdag reg2
651 H reg1
...
4630 S reg4
4631 CNOT reg4 reg3
4632 H reg4
4633 MeasX reg4
4633 MeasX reg0
4633 MeasX reg1
4633 MeasX reg2
4633 MeasX reg3
#EndFunction
```

## B.4 Leaf Schedules

The leaf functions (all modules without submodule calls) are scheduled by the Perl scheduler. Each function is independently evaluated, so all scheduling is fine-grained and no rescheduling of parallel modules is performed. The Perl scheduler handles both RCP and LPFS scheduling. The shown results are for LPFS with $k = 4$, $d = 1024$, $l = 1$, with SIMD and Refill enabled.

Due to length the following is excerpted to show interesting details.

```
M: $::SIMD_K=4; $::SIMD_D=1024; $::SIMD_L=1
LPFS:
Function: main (sched: lpfs, op_cnt: 8426, k: 4, d: 1024, l: 1, opp: 1, refill:↩
     1)
================================================================================↩

ops = 7673
moves = 753
total = 8426
ots = 4633
mts = 561
ts = 6877
SIMDs = 4
tgates = 2459
T(1) = 15346
T(inf) = 4633
T(4,1024) = 6877
Speedup = 2.23149629198779
Efficiency = 0.557874072996946
Utility = 0.306310891377054
Quality = 0.306310891377054
Overhead = 8.93662473296938% (reduction: 57.7300419336973)
Avg load = 1.34224598930481
Peak load = 5

0,0 MOV 1 0 reg0
0,0 MOV 1 0 reg1
0,0 MOV 1 0 reg2
0,0 MOV 1 0 reg3
0,0 MOV 1 0 reg4
0,1 1669: PrepZ reg3
0,1 1: PrepZ reg1
0,1 2941: PrepZ reg4
0,1 343: PrepZ reg0
0,1 651: PrepZ reg2
1,0 MOV 2 1 reg0
1,1 1672: Tdag reg3
1,1 2945: Tdag reg4
1,1 2: Tdag reg1
1,1 653: Tdag reg2
1,2 344: H reg0
2,1 1673: Tdag reg3
2,1 2948: Tdag reg4
2,1 3: Tdag reg1
2,1 656: Tdag reg2
2,2 347: T reg0
3,1 1675: Tdag reg3
3,1 2950: Tdag reg4
3,1 4: Tdag reg1
3,1 657: Tdag reg2
4,1 1677: Tdag reg3
4,1 2953: Tdag reg4
4,1 5: Tdag reg1
4,1 660: Tdag reg2
5,0 MOV 0 2 reg0
5,0 MOV 2 1 reg3
5,0 MOV 3 1 reg4
5,1 662: Tdag reg2
```

```
5,1 6: Tdag reg1
5,2 1680: T reg3
5,3 2956: H reg4
6,1 664: Z reg2
6,1 7: Z reg1
6,2 1682: H reg3
6,3 2959: Tdag reg4
7,1 665: H reg2
7,1 8: H reg1
7,2 1684: Tdag reg3
7,3 2964: Z reg4
...
4630,1 7667: CNOT reg4 reg3
4631,0 MOV 0 2 reg2
4631,0 MOV 2 1 reg3
4631,1 7668: H reg4
4631,2 7673: MeasX reg3
4632,1 7669: MeasX reg4
```

## B.5 Full Schedule

After the leaves are scheduled, an LLVM optimizer pass will read in the resulting schedules and perform coarse-grained scheduling of the hierarchical modules. Only the final results are typically reported.

```
#Function main
SIMD k=4 d=1024 main 4 4633 0 0 1832 1832 4 4 leaf=1

#Num of SIMD time steps for function main : 4633
```

# Appendix C

# Resources

The ScaffCC code can be obtained from GitHub at `https://github.com/ajavadia/ScaffCC`.

The scheduling scripts can be obtained at `https://github.com/jheckey/ScaffSched`.